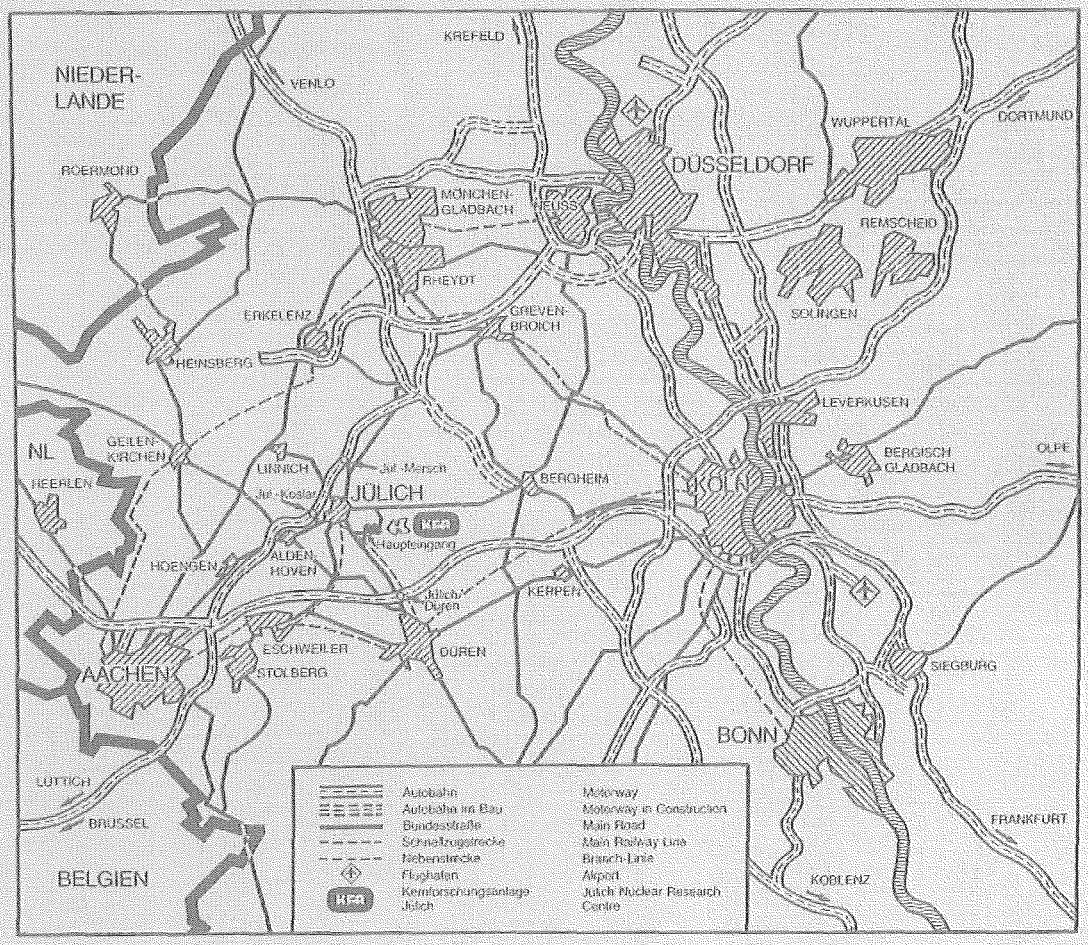
KERNFORSCHUNGSANLAGE JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik

Möglichkeiten des Multitasking zur Beschleunigung von Standardalgorithmen

von

Siegfried Knecht

Jül - Spez - 361
Juli 1986
ISSN 0343-7639



Als Manuskript gedruckt

Spezielle Berichte der Kernforschungsanlage Jülich – Nr. 361

Zentralinstitut für Angewandte Mathematik Jül - Spez - 361

Zu beziehen durch: ZENTRALBIBLIOTHEK der Kernforschungsanlage Jülich GmbH
Postfach 1913 · D-5170 Jülich (Bundesrepublik Deutschland)
Telefon: 02461/610 · Telex: 833556-0 kf d

Möglichkeiten des Multitasking zur Beschleunigung von Standardalgorithmen

von

Siegfried Knecht

Diese Arbeit wurde als Diplomarbeit am Institut für Allgemeine Elektrotechnik und Datenverarbeitungssysteme der Rheinisch-Westfälischen Technischen Hochschule Aachen bei Herrn Prof. Dr.-Ing. O. Lange im Zentralinstitut für Angewandte Mathematik der KFA Jülich angefertigt.

Für die Unterstützung durch das Zentralinstitut für Angewandte Mathematik bedanke ich mich beim Institutsdirektor Herrn Dr. F. Hoßfeld.

1.0 Einleitung 1

2.0 Interaktionen paralleler Prozesse 3
2.1 Präzedenzgraphen 7
2.2 Spezifikation paralleler Ausführung 9
2.2.1 Fork und Join 9
2.2.2 Die parallele Anweisung 11
2.2.3 Vergleich der beiden Konzepte 12
2.3 Verwendung gemeinsamer Variablen 15
2.3.1 Wechselseitiger Ausschluß kritischer Sektionen 15
2.3.2 Semaphore 22
2.3.3 Kritische Bereiche 26
2.3.4 Monitore 30
2.4 Message Passing 32
2.4.1 Kommunikationsarten 33
2.4.2 Synchronisation 35
2.5 Das Deadlock-Problem 37
2.5.1 Vermeidung von Deadlocks 40
2.5.2 Verhinderung von Deadlocks 41
2.5.3 Entdeckung und Beseitigung von Deadlocks 42

3.0 Das Konzept des Multitasking 45
3.1 Speedup durch Multitasking 46
3.1.1 Theoretischer Speedup und Amdahls Gesetz 46
3.1.2 Granularität 50
3.1.3 Balancierte Verteilung paralleler Programmteile 52
3.2 Effizienz, Redundanz und Auslastung eines Systems 55
3.3 Multitasking auf der CRAY X-MP/22 57
3.3.1 Beschreibung der CRAY X-MP/22 Architektur 57
3.3.2 Inter-Prozessor Kommunikationssektion 58
3.3.3 Multitasking-unterstützende System-Software 61
3.3.4 Multitasking-Programmierung 65
3.3.4.1 Taskinitiierung und Taskabschluß 65
3.3.4.2 Synchronisation mit EVENT-Routinen 67

3.3.4.3 Behandlung kritischer Sektionen 69
3.3.4.4 Abstimmung durch den Programmierer 71
3.4 Probleme mit Multitasking 73
3.4.1 Bereiche von Variablen 74
3.4.2 Gefahr eines Deadlocks 75
3.4.3 Overhead der Multitasking-Routinen 76

4.0 Nutzung der Multitasking-Fähigkeiten für Standardalgorithmen 81
4.1 Klassifikation paralleler Algorithmen 82
4.1.1 Asynchrone parallele Algorithmen 83
4.1.1.1 Algorithmus zur Minimum-Maximum-Suche 84
4.1.1.2 Quicksort 88
4.1.2 Synchrone parallele Algorithmen 101
4.1.2.1 Jacobi-Verfahren 102
4.1.2.2 LU-Faktorisierung 111
4.1.2.3 Cholesky-Dekomposition 119

5.0 Schlußbemerkungen 127

Literatur 129

1.0 EINLEITUNG

In den letzten Jahren haben enorme Fortschritte in der Halbleiter- und Schaltkreistechnologie zur Entwicklung von immer leistungsfähigeren von-Neumann-Rechnersystemen geführt. Trotz des dadurch bedingten Leistungszuwachses sind in einem Rechnersystem mit sequentiellem Befehlsstrom einer weiteren Leistungssteigerung durch die physikalischen Restriktionen der Hardware Grenzen gesetzt. Diese Einschränkung sowie die Forderung nach immer höheren Rechnerleistungen und ein drastischer Rückgang der Hardware-Preise führten zur Entwicklung neuer innovativer Rechnerarchitekturen, die aufgrund der Mehrfachausführung ihrer Hardware-Komponenten in der Lage sind, große Mengen von Operationen und Daten parallel zu verarbeiten. Damit zeichnet sich im Design heutiger moderner Rechnersysteme der Trend ab, möglichst viele gleichzeitig arbeitende Hardware-Komponenten zuzulassen. Die Erwartungen, die an diese Parallelrechner geknüpft werden, sind neben einer hohen Zuverlässigleit (reliability) und Verfügbarkeit (availability) eine deutliche Verbesserung der "turnaround"-Zeit und damit eine erhöhte Durchsatzrate.
Da sich mit der Abkehr von der von-Neumann-Rechnerarchitektur auch die Operationsprinzipien und die Organisationstruktur der Rechner verändert haben, bedeutet dies insbesondere die Entwicklung neuer Betriebssyteme, Programmiersprachen und Compiler. Damit die Fähigkeit zur Parallelverarbeitung und das größere Leistungsvermögen dieser Rechnersysteme effektiv genutzt werden kann, wird es nötig sein, originäre parallele Algorithmen neu bzw. weiter zu entwickeln. Voraussetzung für eine Leistungssteigerung ist die Parallelisierbarkeit der zur Lösung anstehenden Probleme. Die Umstrukturierung eines sequentiellen Algorithmus muß gleiche Ergebnisse sicherstellen. Zur Ermittlung der Leistungsverbesserung wird man jeweils den optimalsten parallelen bzw. sequentiellen Algorithmus heranziehen.

In einem Rechnersystem parallel auszuführende Algorithmen erfordern eine Reihe von Mechanismen, die dazu dienen, die auftretenden parallelen Operationen zu koordinieren. Kapitel 2 gibt einen Überblick über die verschiedenen Kommunikations- und Synchronisationsmechanismen, die zur Steuerung paralleler Operationen notwendig sind. Darüberhinaus wird das

sich aus der parallelen Ausführung der Programme und infolge Betriebsmittelknappheit möglicherweise ergebende Deadlock-Problem behandelt.

Die Ausführung paralleler Algorithmen kann mit Hilfe verschiedener parallelverarbeitender Konzepte bewerkstelligt werden. Das Konzept des Multitasking ist eine spezielle Form des Multiprocessing. Von der Firma CRAY RESEARCH für Rechnersysteme der CRAY X-MP Serie entwickelt, können mit seiner Hilfe Algorithmen auf Unterprogrammebene parallel ausgeführt werden. Neben dem Konzept des Multitasking wird in Kapitel 3 das Multiprozessorsystem CRAY X-MP/22 vorgestellt, dessen spezielle Architektur es ermöglicht, Parallelismus auf mehreren Ebenen gleichzeitig auszunutzen. Nach einem Überblick über Multitasking-unterstützende Hardware und Software wird erläutert, wie die Multitasking-Fähigkeiten dieses Rechnersystems genutzt werden können. Gleichzeitig werden die Einflußfaktoren des durch Multitasking erreichbaren Speedups näher erläutert.

Parallele Algorithmen lassen sich in asynchrone und synchrone parallele Algorithmen unterteilen. Während in einem synchronen Algorithmus eine Task auf die Ergebnisse anderer Tasks angewiesen ist, sind die Berechnungen paralleler Tasks in einem asynchronen Algorithmus völlig unabhängig. Daraus ergibt sich die Charakteristik eines ungleichen Aufwandes für Kommunikation und Synchronisation in beiden Arten von Algorithmen, d.h. je nach Problemstellung und -komplexität ein deutlicher Unterschied im Overhead. In Kapitel 4 wird anhand von Standardalgorithmen beider Typen beispielhaft untersucht, welche Beschleunigung durch Nutzung der Multitasking-Fähigkeiten auf einer CRAY X-MP/22 erzielt wird. Im Anschluß an die Parallelisierung eines Algorithmus zur Bestimmung des Minimums und Maximums von N reellen Zahlen wird untersucht, welche Beschleunigung für einen semi-asynchronen Quicksort-Algorithmus erreichbar ist. Daran schließen sich Speedup-Untersuchungen zu drei synchronen Algorithmen an, die der Lösung linearer Gleichungssysteme dienen: das Jacobi-Iterationsverfahren, die LU-Faktorisierung und die Cholesky-Reduktion. Für alle vorgestellten Algorithmen wurde eine der Programmiersprache PASCAL ähnliche Pseudo-Notation verwendet.

2.0 INTERAKTIONEN PARALLELER PROZESSE

Parallelverarbeitung ist eine auf Leistungssteigerung hin ausgerichtete Form der Informationsverarbeitung, die die gleichzeitige Verarbeitung mehrerer Operationen und Daten beinhaltet und damit im Gegensatz zur sequentiellen Verarbeitung steht. Die effektive Steuerung der Aktionen, die sich aus der parallelen Verarbeitung der Prozesse in einem Rechnersystem ergeben, spielt nicht zuletzt wegen der beschränkten Anzahl von Betriebsmitteln, um die diese Prozesse konkurrieren, eine wichtige Rolle.
In den letzten Jahren wurden mehrere höhere parallele Programmiersprachen (ADA, Concurrent PASCAL, CSP, etc.) entwickelt, die es dem Benutzer ermöglichen, die parallele Ausführung mehrerer Programme direkt einzuleiten sowie die Kommunikation und Synchronisation dieser parallel ausgeführten Programme zu steuern. Daneben gibt es mittlerweile einige Preprocessoren, die ein sequentielles Programm in eine parallele Form umstrukturieren. Die so umstrukturierten Programme können, nachdem sie von einem speziellen Compiler übersetzt worden sind, in einem Rechnersystem mit mehreren CPUs effizient ausgeführt werden [GaPe,85].
Die folgenden Problemstellungen spielen bei der Spezifikation einer parallelen Programmiersprache eine wichtige Rolle:

1. Wie wird eine parallele Ausführung spezifiziert?
2. Wie kommunizieren die parallelen **Prozesse**?
3. Wie werden parallele Prozesse synchronisiert?

Ein *Prozeß* spezifiziert die sequentielle Ausführung einer Liste von Anweisungen, d.h. ein Prozeß ist ein Programm(teil) in Ausführung. Dagegen ist ein Programm selbst kein Prozeß: ein Programm ist eine passive Einheit, während ein Prozeß eine aktive Einheit ist.
Ein Prozeß kann sich jeweils in einem der folgenden vier Zustände befinden [PeSi,85]:

- **Aktiv** (Running): Die Anweisungen des Prozesses werden ausgeführt.
- **Blockiert** (Blocked): Der Prozeß wartet auf das Eintreten eines Ereignisses (z.B. Ende einer E/A-Anforderung).

- **Bereit** (Ready): Der Prozeß wartet darauf einem Prozessor zugeteilt zu werden.
- **Verklemmt** (Deadlocked): Der Prozeß wartet auf ein Ereignis, das nicht eintreten wird.

Das Zustandsdiagramm in Abb. 1 entspricht diesen vier Zuständen.

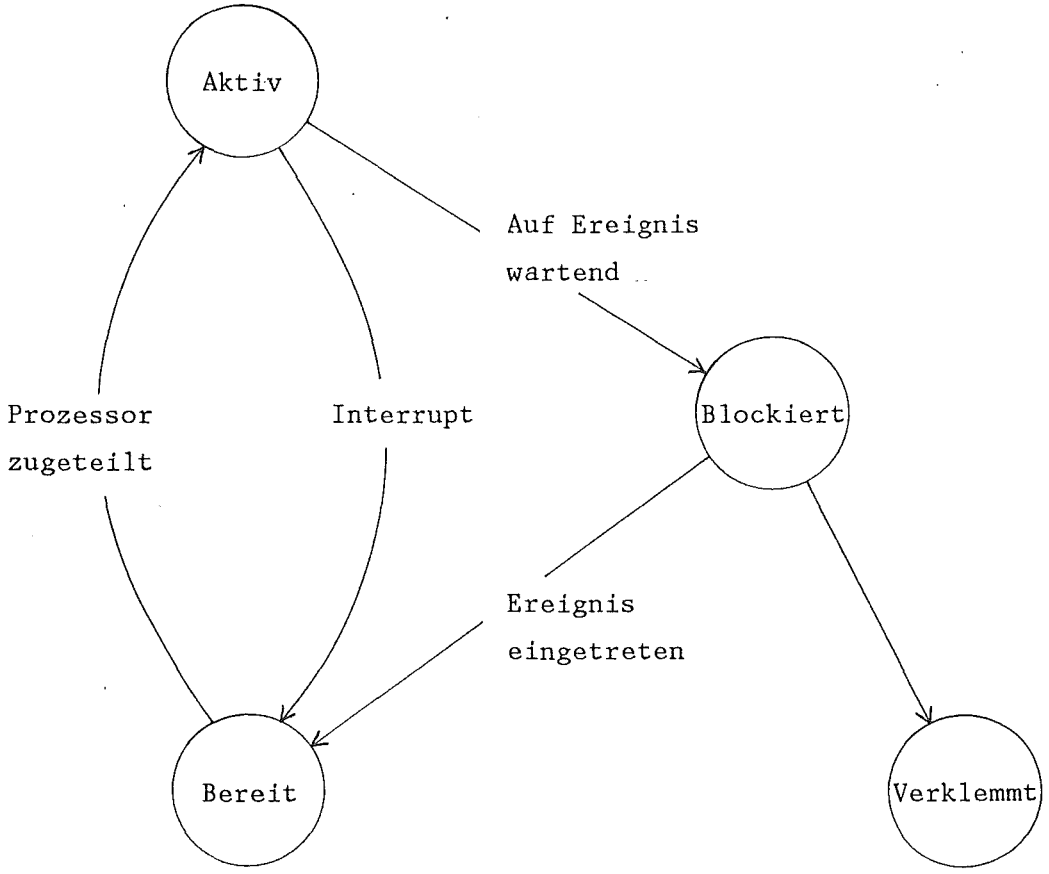


Abb. 1. Zustandsdiagramm eines Prozesses

Parallele Prozesse spezifizieren dagegen zwei oder mehrere Programme, die gleichzeitig ausgeführt werden können.
Die Ausführung paralleler Programme kann mit Hilfe verschiedener Betriebsformen bewerkstelligt werden. Sind für die parallele Ausführung eines Programms mehrere CPUs verfügbar, gibt es zwei Möglichkeiten:

1. Teilen sich die Prozessoren bei der parallelen Ausführung von Programmen einen *gemeinsamen Speicher*, so spricht man von **Multiprocessing**. Jeder der Prozessoren kann zusätzlich einen kleinen lokalen Speicher oder Puffer enthalten. Um die Verbindung zwischen den Prozessoren und dem gemeinsamen Speicher zu bewerkstelligen, bedient man sich entweder eines Verbindungsnetzwerkes, oder der gemeinsame Speicher wird mit mehreren Eingängen ausgestattet. Die Kommunikation zwischen den Prozessoren findet über den gemeinsamen Speicher statt. Man nennt dieses System ein **stark gekoppeltes System** (tightly coupled system).

2. Teilen sich die Prozessoren *keinen gemeinsamen Speicher*, so nennt man diese Art paralleler Ausführung **Distributed Processing**. Jeder Prozessor hat seinen eigenen lokalen Speicher. Der Prozessor, sein lokaler Speicher und seine E/A-Einheiten werden zu einem *Computer-Modul* zusammengefaßt. Prozesse, die auf verschiedenen Computer-Modulen verarbeitet werden, kommunizieren durch Austausch von Nachrichten über ein *Nachrichtenübertragungssystem* (message transfer system). Man spricht in diesem Fall von einem **schwach gekoppelten System** (loosely coupled system) [HwBr,84].

Abb. 2 soll den Sachverhalt bei einem stark bzw. schwach gekoppelten System noch einmal deutlich machen (P bezeichnet Prozessor und CM Computer-Modul).

Die Betriebsform des **Multiprogramming** läßt sich sowohl für Systeme mit mehreren CPUs als auch für Systeme mit nur einer CPU implementieren. Bei Verwendung des Multiprogramming befinden sich mehrere Programme gleichzeitig im Speicher, um eine schnelle Umschaltung zwischen den Prozessen zu gewährleisten. Diese Prozesse greifen gemeinsam auf die Betriebsmittel (CPU, E/A-Subsystem, Speicher, etc.) des Systems zu und ermöglichen dadurch eine günstige Ausnutzung dieser Betriebsmittel [Zima,80].

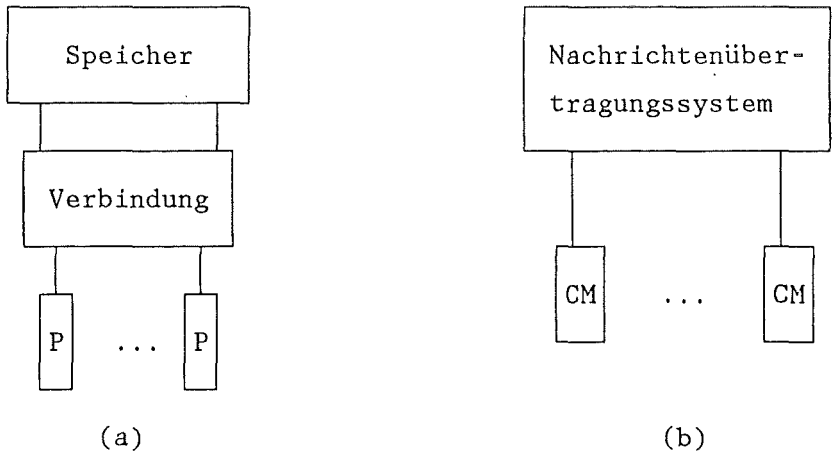


Abb. 2. Multiprozessorsysteme: stark (a), schwach gekoppelt (b).

Bei paralleler Verarbeitung durch mehrere Prozessoren müssen Prozesse miteinander kommunizieren. Mit der Kommunikation eng verbunden ist die Synchronisation der Prozesse, die für eine erfolgreiche Kooperation paralleler Prozesse unumgänglich ist.

Kommunikation erlaubt einem Prozeß einen anderen Prozeß zu beeinflussen. Die Kommunikation zwischen den Prozessen basiert auf der Verwendung *gemeinsamer Variablen* (shared variables: Variablen, auf die mehrere Prozesse zugreifen können) oder auf *Message Passing* [AnSch,83].

Synchronisation ist notwendig, wenn Prozesse miteinander kommunizieren. Zur Synchronisation mit einem anderen Prozeß muß ein Prozeß eine Tätigkeit ausführen, die der andere dann entdeckt - eine Tätigkeit wie das Setzen einer Variablen auf einen bestimmten Wert oder das Senden einer Nachricht. Dies klappt aber nur dann, wenn die Ereignisse "führe eine Tätigkeit aus" und "entdecke eine Tätigkeit" gezwungen werden, in dieser Reihenfolge stattzufinden. Synchronisation kann deshalb als eine Menge von Restriktionen an die Reihenfolge des Eintretens von Ereignissen gesehen werden. Damit solche Restriktionen auch gewährleistet sind, bedient sich ein Programmierer der ihm zur Verfügung stehenden Synchronisationsmechanismen, um die Prozesse zu steuern.

Es stellt sich nun die Frage, wie diese Restriktionen anschaulich dargestellt werden können und wann die parallele Ausführung verschiedener Anweisungsfolgen möglich ist.

2.1 PRÄZEDENZGRAPHEN

Die Ausführungsreihenfolge verschiedener Anweisungen in einem Programm ist an bestimmte Restriktionen gebunden. Ein arithmetischer Ausdruck kann z.B. erst dann ausgewertet werden, wenn die für die Auswertung benötigten Daten auch vorliegen.
Diese Restriktionen können anhand eines **Präzedenzgraphen** dargestellt werden [PeSi,85]. Daneben ist auch die Darstellung durch eine *Präzedenzmatrix* möglich [KNU,75].

Definition 2.1

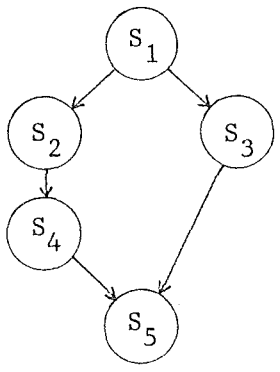
Ein *Präzedenzgraph* ist ein gerichteter, azyklischer Graph G = (V,E), dessen Knoten $v \in V$ einzelnen Anweisungen entsprechen. Eine Kante $e_{ij} \in E$ von Knoten v_i zu Knoten v_j bedeutet, daß Anweisung S_j erst dann ausgeführt werden kann, wenn die Ausführung der Anweisung S_i beendet ist.

Beispiel 2.1 zeigt ein Programmsegment, das Information von einer E/A-Einheit liest und dann Information auf eine zweite E/A-Einheit schreibt, sowie den zugehörigen Präzedenzgraphen G.

Beispiel 2.1

```
S1:   read(input1,a)
S2:   b := a + 2
S3:   c := a - 1
S4:   d := a + b
S5:   write(output1,c+d)
```

G:



Es ist klar, daß in Beispiel 2.1 die Anweisungen S_2 und S_3 erst dann ausgeführt werden können, wenn die Anweisung S_1 abgeschlossen ist. Ebenso kann S_4 nicht ausgeführt werden, bevor der Wert von b berechnet worden

ist. Auf der anderen Seite jedoch zeigt der Präzedenzgraph, daß die Anweisung S_2 gefolgt von S_4 parallel zu Anweisung S_3 ausgeführt werden kann. Es gilt also zu überprüfen, wann verschiedene Anweisungen parallel ausgeführt werden können und trotzdem die gleichen Ergebnisse liefern. Bevor eine Antwort auf diese Frage gegeben werden kann, seien die folgenden Bezeichnungen eingeführt.

- $EIN(S_i) = \{a_1, a_2, \ldots, a_{mi}\}$, die *Eingabemenge* von S_i, ist die Menge aller Variablen aus S_i, deren Werte während der Ausführung von S_i benutzt werden.
- $AUS(S_i) = \{b_1, b_2, \ldots, b_{ni}\}$, die *Ausgabemenge* von S_i, ist die Menge aller Variablen aus S_i, deren Werte während der Ausführung von S_i geändert werden.

Für die Anweisung c := a + b sehen Ein- und Ausgabemenge wie folgt aus:

$$EIN(c := a + b) = \{a,b\}$$
$$AUS(c := a + b) = \{c\}$$

Der Durchschnitt beider Mengen muß nicht die leere Menge sein. Beispielsweise ist für die Anweisung a := a + 1

$$EIN(a := a + 1) = AUS(a := a + 1) = \{a\}.$$

Die folgenden drei Bedingungen, in der Literatur als **Bernstein-Bedingungen** bezeichnet [HwBr,84], müssen erfüllt sein, damit zwei Anweisungen S_1 und S_2 parallel ausgeführt werden können und trotzdem dasselbe Ergebnis liefern:

1. $EIN(S_1) \cap AUS(S_2) = \emptyset$
2. $AUS(S_1) \cap EIN(S_2) = \emptyset$
3. $AUS(S_1) \cap AUS(S_2) = \emptyset$

Die Anweisungen S_2 und S_3 aus Beispiel 2.1 können deshalb parallel ausgeführt werden, weil

$$EIN(S_2) = \{a\}, \quad AUS(S_2) = \{b\},$$
$$EIN(S_3) = \{a\}, \quad AUS(S_3) = \{c\}$$

ist und damit die Bernstein-Bedingungen erfüllt sind. Demgegenüber können S_2 und S_4 nicht parallel ausgeführt werden, weil folgendes gilt:

$$AUS(S_2) \cap EIN(S_4) = \{b\}.$$

Nachdem die Bedingungen für die parallele Ausführung verschiedener Programmteile vorgestellt worden sind, sollen nun einige Konzepte betrachtet werden, mit deren Hilfe man parallele Ausführung initiieren kann.

2.2 SPEZIFIKATION PARALLELER AUSFÜHRUNG

Obgleich Präzedenzgraphen ein nützliches Mittel sind, um Restriktionen, an die einzelne Teile einer Berechnung gebunden sind, zu beschreiben, lassen sie sich als zweidimensionales Objekt nur schwer in Programmiersprachen einbauen. Deshalb werden andere Mittel benötigt, die sowohl die Präzedenzrelationen als auch die parallele Ausführung spezifizieren.

2.2.1 Fork und Join

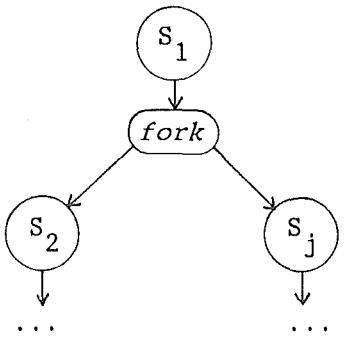
Die Anweisungen **fork** und **join** wurden Mitte der sechziger Jahre eingeführt und gehören zu den ersten programmiersprachlichen Notationen, um Parallelität auszudrücken [PeSi,85].
Die "**fork** L"-Anweisung initiiert zwei parallele Ausführungen in einem Programm. Die erste Ausführung startet bei der Anweisung mit Label L, während die zweite bei der Anweisung beginnt, die auf die "**fork** L"-Anweisung folgt.

Beispiel 2.2

Dieses Beispiel zeigt die Verwendung der "**fork** L"-Anweisung und den dazugehörigen Präzedenzgraphen G_f. Die erste Ausführung startet bei der Anweisung S_j und die zweite bei der Anweisung S_2.

```
    S1
    fork L
    S2
    ...
L:  Sj
    ...
```

G_f:



Die **join**-Anweisung ist das Mittel, um zwei parallele Berechnungen wieder zu einer zusammenzufügen. Jede der beiden Berechnungen muß die **join**-Anweisung enthalten. Weil die Berechnungen mit unterschiedlicher Geschwindigkeit ablaufen können, wird jene, die zuerst **join** ausführt, beendet, während die andere weitermachen kann.
Das Betriebssystem UNIX [RiTh,74] macht umfassend Gebrauch vom **fork/join**-Konzept: **fork** dient der Erzeugung von Prozessen, während eine Variante von **join** die Prozesse nach deren Beendigung wieder zusammenfügt.

Obwohl die **fork**-Anweisung einen direkten Mechanismus zur dynamischen Prozeßerstellung schafft, haben **fork** und **join** eine wenig attraktive Eigenschaft. Programme, die sich dieser Anweisungen bedienen, haben eine unhandliche Kontrollstruktur, denn eine **fork**-Anweisung ist in ihrer Ausführung der **goto**-Anweisung ähnlich. Die **goto**-Anweisung ist unattraktiv, weil sie den Grundsätzen, die durch die strukturierte Programmierung eingeführt worden sind, widerspricht.

2.2.2 Die parallele Anweisung

Ein anderes, besser strukturiertes Konzept zur Spezifikation paralleler Ausführung ist die **parbegin/parend**-Anweisung von Dijkstra [PeSi,85], die folgendes Aussehen hat:

parbegin S_1; S_2; ...; S_n **parend**

Jedes der S_i stellt eine Anweisung dar. Alle Anweisungen zwischen **parbegin** und **parend** können parallel ausgeführt werden. Dabei können mehrere Anweisungen durch **begin** und **end** zu einer parallel auszuführenden Anweisung geklammert werden. Abb. 3 zeigt den Präzedenzgraphen G_p dieses Konzepts. In der Literatur wird diese Anweisung auch häufig mit **cobegin/coend** bezeichnet [Holt,78], [Ari,82].

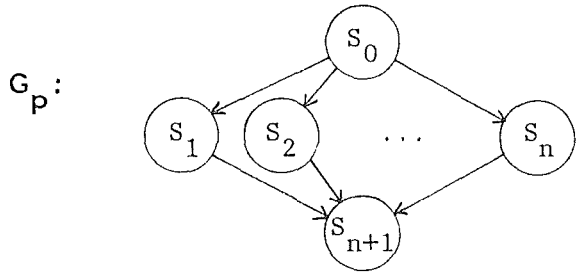


Abb. 3. Präzedenzgraph der parallelen Anweisung.

Eine der Anweisungen, die auf **parbegin/parend** folgt (S_{n+1} in G_p), kann erst dann ausgeführt werden, wenn die Ausführungen *aller* Anweisungen zwischen **parbegin** und **parend** abgeschlossen sind.
Beispiel 2.1 sieht mit Verwendung der **parbegin/parend**-Anweisung folgendermaßen aus:

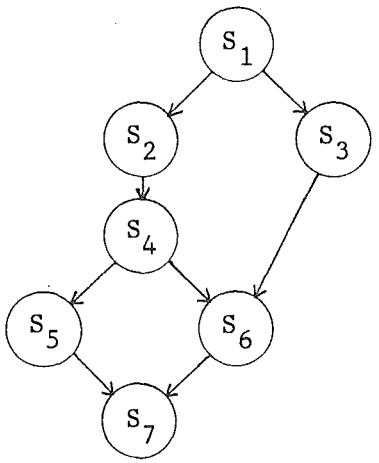
```
read(input1,a)
parbegin
   c := a - 1
   begin
      b := a + 2
      d := a + b
   end
parend
write(output1,c+d)
```

Die parallele Anweisung **parbegin/parend** läßt sich wegen ihrer Übersichtlichkeit leicht in moderne, blockstrukturierte höhere Programmiersprachen integrieren. Eine Variante davon ist in den Programmiersprachen EDISON [BrHa,81] und ALGOL68 [Wijn,75] implementiert.

2.2.3 Vergleich der beiden Konzepte

Da sowohl **parbegin/parend** als auch **fork/join** die anhand eines Präzedenzgraphen dargestellten Präzedenzrelationen ausdrücken sollen, stellt sich die Frage, ob beide Konzepte geeignet sind, jeden möglichen Präzedenzgraphen zu modellieren. Dazu betrachte man den folgenden Graphen G_1:

G_1:



Dieser Präzedenzgraph kann nicht durch ein Programm, welches nur die **parbegin/parend**-Anweisung benutzt, ausgedrückt werden. S_6 kann nur dann

parallel mit S_5 ausgeführt werden, wenn die Ausführung von S_3 beendet ist. Da über die Ausführungsgeschwindigkeit der Prozesse keine Annahme gemacht werden kann, läßt sich nicht mit Sicherheit vorherbestimmen, daß S_3 tatsächlich beendet ist.

Die parallele Anweisung **parbegin** S_1; S_2; ...; S_n **parend** läßt sich jedoch auf recht einfache Weise durch **fork/join** simulieren:

```
        Zähler := n
        fork L2
        fork L3
          ...
        fork Ln
        S1
        goto Lm
L2:     S2
        goto Lm
L3:     S3
        goto Lm
          ...
Ln:     Sn
Lm:     join(Zähler)
```

Hier taucht das Problem auf, daß man die Anzahl der zu vereinigenden Berechnungen kennen muß, damit alle bis auf die letzte beendet werden können. Sollen beispielsweise drei Berechnungen vereinigt werden, dann werden die ersten beiden, die **join** ausführen, beendet, während die dritte weitermachen kann. Um dies zu bewerkstelligen, wird die **join**-Anweisung um einen Parameter erweitert, der die Anzahl zu vereinigender Berechnungen enthält. Die Ausführung einer **join**-Anweisung mit dem Parameter *Zähler* hat folgenden Effekt:

```
Zähler := Zähler - 1
if Zähler > 0 then einstellen
```

Dabei ist *Zähler* eine nicht-negative ganzzahlige Variable und *einstellen* eine Instruktion, die zur Beendigung der Aktivität eines Prozesses führt.

Damit ist klar, daß mit Hilfe von **fork/join** der Graph G_1 ohne weiteres zu modellieren ist:

```
        S1
        Zähler1 := 2
        fork L1
        S2
        S4
        Zähler2 = 2
        fork L2
        S5
        goto L3
L1:     S3
L2:     join(Zähler1)
        S6
L3:     join(Zähler2)
        S7
```

Obwohl man mit **fork/join** jeden Präzedenzgraphen modellieren kann, zeigt sich schon an diesem noch recht kleinen Problem die Unübersichtlichkeit dieses Konzeptes.

Da über die Ausführungsgeschwindigkeit der durch die **fork**-Anweisung erzeugten parallelen Prozesse nichts ausgesagt werden kann, könnten unter Umständen mehrere Prozesse die **join**-Anweisung gleichzeitig ausführen, und damit gleichzeitig auf die gemeinsame Variable *Zähler* zugreifen. Damit es nicht zu solchen Zugriffskonflikten kommen kann, muß die **join**-Anweisung als **atomare Operation** ausgeführt werden.

Eine *atomare* (unteilbare) *Operation* ist eine Operation, die nicht unterbrochen werden kann. Wurde ihre Ausführung erst einmal gestartet, ist sichergestellt, daß ein anderer Prozeß die vollständige Ausführung dieser Operation nicht stören und damit ihr Ergebnis nicht beeinflußen kann.

Zugriff und Test auf die gemeinsame Variable Zähler dürfen also immer nur von einem Prozeß ausgeführt werden. Alle anderen Prozesse müssen während dieser Zeit verzögert werden. Dieser Sachverhalt führt hin zum Problem der Synchronisation mehrerer Prozesse durch die Verwendung gemeinsamer Variablen.

2.3 VERWENDUNG GEMEINSAMER VARIABLEN

Wie im letzten Abschnitt gesehen, kann es während der Ausführung paralleler Programme vorkommen, daß mehrere Prozesse gemeinsame Variablen gleichzeitig benutzen möchten oder daß ein Prozeß das Ergebnis einer Berechnung eines anderen Prozesses für seine Berechnungen benötigt. Um zu verhindern, daß mehrere Prozesse gleichzeitig auf eine Variable zugreifen und um sicherzustellen, daß das Ergebnis einer Berechnung auch vorliegt, wenn es von einem anderen Prozeß benötigt wird, müssen die Prozesse kommunizieren und synchronisieren.

Die Synchronisation anhand gemeinsamer Variablen erfordert den zeitweisen exklusiven Zugriff eines Prozesses auf diese gemeinsame Variable. Diese Forderung führt zum Problem des **wechselseitigen Ausschlusses** der Ausführung jener Programmabschnitte, in denen mehrere Prozesse gemeinsame Variablen beanspruchen. Diese Bereiche werden als **kritische Sektionen** bezeichnet.

2.3.1 Wechselseitiger Ausschluß kritischer Sektionen

Gegeben sei ein System von n kooperierenden Prozessen $\{P_1, P_2, \ldots, P_n\}$. Jeder Prozeß besitze eine *kritische Sektion*, in welcher er ein exklusives Betriebsmittel beansprucht. Insbesondere bedeutet dies, daß ein Prozeß in seiner kritischen Sektion eine von mehreren Prozessen gemeinsam benutzte Variable lesen und/oder überschreiben kann. Die Ausführung kritischer Sektionen ist *wechselseitig ausgeschlossen*, d.h. werden in der kritischen Sektion eines Prozesses dieses Systems Anweisungen ausgeführt, können keine Anweisungen in der kritischen Sektion eines anderen Prozesses dieses Systems ausgeführt werden.
Das Problem der kritischen Sektion besteht darin, ein Protokoll zu entwerfen, welches die Prozesse zur Kooperation benutzen müssen. Jeder Prozeß fragt nach der Erlaubnis seine kritische Sektion betreten zu dürfen. Diese Anfrage geschieht durch ein **Eintritt**-Protokoll. Gefolgt wird die kritische

Sektion von einem Austritt-Protokoll, wodurch das Verlassen der kritischen Sektion angezeigt wird [PeSi,85].
Eine Lösung des Problems des wechselseitigen Ausschlusses sollte folgende Anforderungen erfüllen [KNU,75]:

1. Zu jedem Zeitpunkt darf sich höchstens einer der Prozesse in der kritischen Sektion aufhalten (wechselseitiger Ausschluß).

2. Befindet sich ein Prozeß in seiner nichtkritischen Sektion, wo er beispielsweise durch Prozessorentzug angehalten wird, kann er die Aktivitäten anderer Prozesse nicht beeinflussen.

3. Versuchen mehrere Prozesse gleichzeitig in ihre kritische Sektion einzutreten, muß einem davon innerhalb endlicher Zeit der Eintritt gewährt werden (Deadlock-Freiheit).

4. Ein Prozeß, der in seine kritische Sektion einzutreten versucht, darf vom Eintritt nicht beliebig lange ausgeschlossen werden (Fairneß).

5. Die oben genannten Anforderungen 1. - 4. dürfen nicht von der Ausführungsgeschwindigkeit der einzelnen Prozesse abhängen.

Der hauptsächliche Unterschied aller Lösungen des Problems des wechselseitigen Ausschlusses besteht darin, wie einfach oder komplex die atomaren Operationen der einzelnen Lösungen gestaltet sind.
Die Lösungen des Problems des wechselseitigen Ausschlusses sind insofern keine tatsächlichen Lösungen, weil durch die Definition atomarer Operationen das Problem auf eine niedrigere, hardware-nähere Ebene verlagert wurde. Eine atomare Operation kann nämlich selbst wieder als exklusives Betriebsmittel angesehen werden. Eine Lösung des Problems des wechselseitigen Ausschlusses wird also nur auf einer höheren Ebene, beispielsweise in einer höheren Programmiersprache, angeboten, wodurch der Benutzer der Sprache den wechselseitigen Ausschluß kontrollieren kann.

Der folgende Algorithmus ist eine Vereinfachung des **Dekker**-Algorithmus [Ari,82] und stellt ebenfalls eine Lösung des Problems des wechselseitigen Ausschlusses für *zwei* Prozesse vor [Pet,81].

Der Algorithmus benötigt drei gemeinsame Variablen:

- Die logische Variable *ein1* bzw. *ein2* ist wahr, wenn Prozeß P_1 bzw. P_2 sich in seinem **Eintritt**-Protokoll befindet oder seine kritische Sektion ausführt.

- Die Variable *nummer* enthält den Namen (1 oder 2) des Prozesses, dem als nächstem der Eintritt in seine kritische Sektion erlaubt wird; *nummer* ist dann nötig, wenn beide Prozesse ihre **Eintritt**-Protokolle gleichzeitig ausführen (*warte* ist eine dummy-Instruktion).

- Alle Zuweisungen in den **Eintritt**- und **Austritt**-Protokollen werden atomar ausgeführt.

```
Algorithmus wechselseitiger_Ausschluß:
var nummer : 1..2
    ein1,ein2 : boolean
begin
   ein1     := false
   ein2     := false
   nummer   := 1              {Initialisierung mit 1 oder 2 ist beliebig}
   parbegin
      process P1
         repeat
            {Eintritt-Protokoll}
               ein1 := true          {Absicht einzutreten}
               nummer := 2           {Priorität auf anderen Prozeß setzen}
               while ein2 and nummer = 2 do
                  warte              {warte, wenn ein anderer Prozeß
                                          versucht und an der Reihe ist}
            kritische Sektion
            {Austritt-Protokoll}
               ein1 := false         {Aufgabe der Absicht einzutreten}
            nichtkritische Sektion
         until Bedingung
      process P2
         repeat
            {Eintritt-Protokoll}
               ein2 := true          {Absicht einzutreten}
               nummer := 1           {Priorität auf anderen Prozeß setzen}
                while ein1 and nummer = 1 do
                  warte              {warte, wenn ein anderer Prozeß
                                          versucht und an der Reihe ist}
            kritische Sektion
            {Austritt-Protokoll}
               ein2 := false         {Aufgabe der Absicht einzutreten}
            nicht-kritische Sektion
         until Bedingung
   parend
end.
```

Behauptung:

Der Algorithmus *wechselseitiger_Ausschluß* erfüllt die Anforderungen 1. - 5.

Beweis:

1. *Wechselseitiger Ausschluß*:

   Angenommen P_1 befindet sich in seiner kritischen Sektion. Daraus ergibt sich: *ein1* = true. Wenn nun P_2 sein **Eintritt**-Protokoll ausführt, setzt er *nummer* := 1.

   Weil *nummer* nicht mehr verändert werden kann, solange P_1 sich in seiner kritischen Sektion befindet, ergibt sich für den logischen Ausdruck in der **while**-Schleife von Prozeß P_2:

   (*ein1* **and** *nummer* = 1) ≡ true.

   Somit kann P_2 seine kritische Sektion nicht betreten, und wechselseitiger Ausschluß ist gewährleistet (analog, wenn P_2 sich anfangs in seiner kritischen Sektion befindet).

   Sind beide Prozesse gleichzeitig in ihrem **Eintritt**-Protokoll, so entscheidet der Wert von *nummer*, welcher Prozeß in seine kritische Sektion eintreten darf. Der andere Prozeß muß dann in seiner **while**-Schleife warten, wodurch wechselseitiger Ausschluß gegeben ist.

2. Wird P_1 in seiner nichtkritischen Sektion angehalten, kann er, wegen *ein1* = false, die Arbeit von Prozeß P_2 nicht beeinflussen (analoge Argumentation für P_2).

3. *Deadlock-Freiheit*:

   Angenommen P_1 wartet in seiner **while**-Schleife. Dann wird P_2 innerhalb endlicher Zeit eines der folgenden drei Dinge tun:

   a) P_2 versucht nicht in seine kritische Sektion einzutreten. Damit gilt *ein2* = false, und P_1 kann aktiv werden.

   b) P_2 wartet ebenfalls in seinem **Eintritt**-Protokoll. Das jedoch ist nicht möglich, weil *nummer* entweder 1 oder 2 ist, und somit einer der Prozesse weiterarbeiten kann.

   c) P_2 durchläuft wiederholt seine Protokolle. Dadurch aber setzt er *nummer* := 1 (*nummer* kann, weil P_1 wartet, nicht auf 2 gändert werden), und Prozeß P_1 kann mit seiner Arbeit fortfahren.

4. *Fairneß* ist deshalb gegeben, weil P_1 nur für eine Ausführung der kritischen Sektion von P_2 verzögert wird.

5. Anforderung 5 folgt direkt aus der Beweisführung 1. - 4., in der keinerlei Forderungen an die Ausführungsgeschwindigkeiten der beiden Prozesse gestellt wurden.

□

Nach diesem Algorithmus, der den wechselseitigen Ausschluß für zwei Prozesse sicherstellt, soll nun ein weiterer Algorithmus betrachtet werden, der das Problem des wechselseitigen Ausschlusses für *n* Prozesse löst. Der Algorithmus stammt von Lamport [Lamp,74] und wurde für ein verteiltes System entwickelt, wo jeder der Prozessoren seinen eigenen Speicher hat. Jeder Prozessor kann aus dem Speicher aller anderen Prozessoren lesen. Die gelesene Information braucht er jedoch nur in seinen eigenen Speicher zu schreiben. Der Algorithmus basiert auf einem Verfahren wie es in Bäckereien und ähnlichen Läden angewandt wird (daher auch der Name **bakery algorithm**).
Wenn ein Kunde in den Laden kommt, empfängt er eine Nummer. Der Kunde mit der niedrigsten Nummer wird als nächster bedient. Dieser Algorithmus kann jedoch nicht garantieren, daß zwei Prozesse (Kunden) nicht die gleiche Nummer empfangen. Wenn P_i und P_j die gleiche Nummer erhalten und $i < j$ ist, dann wird zuerst P_i bedient. Die Prozeßnamen sind total geordnet und werden nur einmal vergeben.
Die gemeinsamen Datenstrukturen für alle Prozesse sind:

```
var wähle  : array[0..n-1] of boolean
    nummer : array[0..n-1] of integer
```

Diese werden mit *false* bzw. *0* initialisiert.
Zusätzlich benötigt man noch die folgenden Notationen.

Definition 2.2

Seien a, b, c, d und a_i (i = 0, ..., n-1) natürliche Zahlen.

1. (a,b) < (c,d), wenn a < c oder wenn a = c und b < d ist.
2. $\max(a_0, a_1, \ldots, a_{n-1})$ ist die natürliche Zahl k, so daß $k \geq a_i$ für i = 0, ..., n-1.

Die Struktur von Prozeß P_i sieht folgendermaßen aus:

```
repeat
   {Eintritt-Protokoll}
      wähle[i] := true
      nummer[i] := max(nummer[0], nummer[1], ..., nummer[n-1]) + 1
      wähle[i] := false
      for j := 0 to n-1 do
         begin
            while wähle[j] do warte
            while nummer[j] ≠ 0 and
               (nummer[j],j) < (nummer[i],i) do warte
         end
   kritische Sektion
   {Austritt-Protokoll}
      nummer[i] := 0
   nichtkritische Sektion
until Bedingung
```

Der Beweis, daß auch dieser Algorithmus die Anforderungen 1. - 5. erfüllt, ist [Lamp,74] zu entnehmen.

Nach diesen beiden Algorithmen soll nun noch kurz gezeigt werden, wie sich wechselseitiger Ausschluß auf Hardware-Ebene lösen läßt.

Lösung auf Instruktionsebene

Viele Rechner sehen spezielle Hardware-Instruktionen vor, die es möglich machen, eine Variable in einem *lesen-ändern-schreiben* Speicherzyklus zu testen und zu verändern. Da die Operation *lesen-ändern-schreiben* atomar ausgeführt wird, können solche Instruktionen verwendet werden, um das Problem des wechselseitigen Ausschlusses kritischer Sektionen zu lösen.
Die IBM Rechner der Serie 360/370 enthalten eine solche **Test-and-Set**-Instruktion [Ari,82]. Diese Test-and-Set-Instruktion ist auch im Rechnersystem CRAY X-MP/22 enthalten.

Allgemein kann *Test-and-Set* wie folgt definiert werden:

```
function Test_and_Set (var x : 0..1) : 0..1
   begin
      Test_and_Set := x
      x := 1
   end
```

Mit einer gemeinsamen Variablen *c* (c steht für "condition code"), die mit 0 initialisiert wird, läßt sich wechselseitiger Ausschluß für n Prozesse folgendermaßen implementieren (bei c = 0 ist der Eintritt in die kritische Sektion frei, während sich bei c = 1 ein Prozeß in seiner kritischen Sektion befindet):

```
repeat
  {Eintritt-Protokoll}
     while Test_and_Set(c) > 0 do warte
  kritische Sektion
  {Austritt-Protokoll}
     c := 0
  nichtkritische Sektion
until Bedingung
```

2.3.2 Semaphore

Ein Konzept, das ähnlich einfach wie *Test-and-Set* zu implementieren ist, jedoch einen wesentlich größeren Einsatzbereich bietet, ist das Konzept der **Semaphore**.
Dieser Synchronisationsmechanismus wurde von Dijkstra eingeführt und basiert auf der Verwendung einer gemeinsamen Variablen, genannt *Semaphor* [PeSi,85].

Definition 2.3

Ein *Semaphor S* ist eine ganzzahlige Variable, auf die *nur* durch die beiden *atomaren Standardoperationen* P und V zugegriffen werden kann.

P(*S*): **while** *S* ≤ 0 **do** *warte*
S := *S* - 1

V(*S*): *S* := *S* + 1

Der Begriff "atomare Operation" besagt hierbei folgendes:

- Während ein Prozeß auf ein Semaphor zugreift, kann kein anderer Prozeß gleichzeitig auf dasselbe Semaphor zugreifen.
- Versuchen zwei Prozesse P(*S*) oder V(*S*) gleichzeitig auszuführen, werden die Operationen in beliebiger Reihenfolge sequentiell ausgeführt.

Ein Semaphor *S* wird üblicherweise mit 1 initialisiert. Wenn *S* nur die Werte 0 oder 1 haben kann, nennt man es ein *binäres Semaphor*, weil es wie ein Sperrbit arbeitet, welches nur einem Prozeß erlaubt zu einem bestimmten Zeitpunkt in seiner kritischen Sektion zu sein. Ist *S* = 1, dann ist der Eintritt in die kritische Sektion frei. *S* = 0 zeigt an, daß ein Prozeß sich in seiner kritischen Sektion befindet.
Kann *S* irgendeinen ganzzahligen Wert annehmen, bezeichnet man es als *allgemeines Semaphor* [HwBr,84].

Semaphore können zur Lösung verschiedener Synchronisationsprobleme verwendet werden. Mit ihrer Hilfe läßt sich zum Beispiel das Problem des wechselseitigen Ausschlusses für n Prozesse folgendermaßen lösen.

Die n Prozesse teilen ein gemeinsames binäres Semaphor *S*, welches mit 1 initialisiert wird. Jeder der Prozesse P_i, i = 1, 2, ..., n, ist wie folgt aufgebaut:

```
repeat
   {Eintritt-Protokoll}
      P(S)
   kritische Sektion
   {Austritt-Protokoll}
      V(S)
   nichtkritische Sektion
until Bedingung
```

Diese Lösung des wechselseitigen Ausschlusses kritischer Sektionen (und damit auch die obige Semaphordefinition) hat den Nachteil, daß sie **busy-waiting** erfordert.

Busy-waiting ist ein Nachteil aller bisheriger Lösungen des wechselseitigen Ausschlusses kritischer Sektionen und besagt folgendes: Während ein Prozeß sich in seiner kritischen Sektion befindet, muß jeder andere Prozeß, der versucht seine kritische Sektion zu betreten, im **Eintritt**-Protokoll wiederholt *rotieren* (spinning process) und die Eintritt-Bedingung abfragen. Variablen, die dafür benutzt werden, nennt man *spin locks*.

Als nachteilig erweist sich beim busy-waiting vor allem, daß während des Wartens eines Prozesses ein Prozessor belegt ist und deshalb Prozessorzyklen verschwendet werden [Ari,82].

Um die Notwendigkeit des busy-waiting zu umgehen, kann man die Operationen P und V modifizieren. Hat ein Prozeß die P-Operation ausgeführt und festgestellt, daß $S \leq 0$ ist, dann muß er nach der oben stehenden Lösung aktiv warten. Besser als busy-waiting wäre jedoch, der Prozeß blockiert sich selbst. Befindet er sich im Blockiertzustand, teilt er dies dem CPU-Scheduler mit, der dann einen anderen Prozeß aus einer Bereit-Queue zur Ausführung auswählen kann.

Ein blockierter Prozeß, der wegen eines Semaphors S wartet, wird infolge der Ausführung einer V-Operation durch einen anderen Prozeß wieder gestartet. Der Wiederstart geschieht durch eine sogenannte *wecke-auf*-Operation, welche den Prozeßzustand von blockiert nach bereit ändert und ihn zur Bereit-Queue bringt [PeSi,85]. Das Semaphor hat dann folgendes Aussehen:

```
type semaphor = record
                    Wert  : integer
                    Liste : list of Prozeß
                end
```

Muß ein Prozeß wegen eines Semaphors warten, wird er zur Liste der wartenden Prozesse hinzugefügt. Eine V-Operation entfernt einen Prozeß von der nicht-leeren Liste der wartenden Prozesse und weckt ihn auf. Mit einem Semaphor *S* von diesem Typ können die Semaphoroperationen wie folgt definiert werden.

Definition 2.4

```
P(S):  S.Wert := S.Wert - 1
       if S.Wert < 0 then
          begin
             füge diesen Prozeß zu S.Liste hinzu
             blockiere
          end

V(S):  S.Wert := S.Wert + 1
       if S.Wert ≤ 0 then
          begin
             entferne einen Prozeß P von S.Liste
             wecke-auf(P)
          end
```

Die *blockiere*-Operation suspendiert den Prozeß, der sie auslöst. Die *wecke-auf*-Operation bereitet die Wiederausführung eines blockierten Prozesses P vor.
Im Gegensatz zum binären Semaphor kann das Semaphor *S* hier negative Werte annehmen. Der negierte Wert von *S.Wert* gibt die Anzahl der wartenden Prozesse an.

Aufgrund der Tatsache, daß der Synchronisationsmechanismus durch Semaphore nicht ausreichend strukturiert ist und somit größere Programme

schnell unübersichtlich werden können, muß immer darauf geachtet werden, daß vor dem Eintritt in die kritische Sektion eine P-Operation und beim Austritt eine V-Operation ausgeführt wird. Wird diese Reihenfolge nicht eingehalten, kann das, wie in anderen Fällen der Programmierung auch, unvorhersehbare Effekte zur Folge haben.

Angenommen ein Prozeß vertauscht die Operationen P und V auf einem Semaphor *S*. Durch dieses Vertauschen können mehrere Prozesse gleichzeitig in ihre kritische Sektion eintreten. Damit ist die Forderung nach wechselseitigem Ausschluß verletzt, und es können zeitabhängige Fehler auftreten.

2.3.3 Kritische Bereiche

Um die aufgezeigten Schwierigkeiten bei der Lösung des Problems des wechselseitigen Ausschlusses durch Semaphore zu umgehen, hat man ein neues Konstrukt, den **kritischen Bereich**, eingeführt [BrHa,77a].

Eine Variable *v* vom Typ *T*, die von mehreren Prozessen gemeinsam benutzt werden soll, kann nur innerhalb einer *Bereichsanweisung* der folgenden Form angesprochen werden:

region *v* **do** *S*

Diese Bereichsanweisung besagt: Während die Anweisung *S* ausgeführt wird, kann kein anderer Prozeß auf den Bereich *v* zugreifen, d.h. *S* wird als kritische Sektion ausgeführt (*S* kann auch eine Folge von Anweisungen sein). Werden beispielsweise die beiden Anweisungen

region *v* **do** S_1
region *v* **do** S_2

gleichzeitig in verschiedenen Prozessen ausgeführt, wird das Ergebnis äquivalent zur sequentiellen Ausführung "S_1 gefolgt von S_2" oder "S_2 gefolgt von S_1" sein. Die Ausführungsreihenfolge ist also nicht vorherbestimmt und hängt von der Ausführungsgeschwindigkeit der Prozesse ab [PeSi,85].

Das Konstrukt des kritischen Bereichs schützt vor einfachen Fehlern wie sie bei der Verwendung von Semaphoren gemacht werden konnten. Eine korrekte Abwicklung der Synchronisation soll dadurch erzwungen werden, daß die Aufrufe von P und V nicht explizit geschrieben werden müssen, sondern automatisch erzeugt werden. Dies erfordert jedoch den Einsatz spezieller Compiler oder Preprocessoren.
Für eine Implementierung des kritischen Bereichs wird beispielsweise für jede Deklaration

    var *v* : *T*

ein Semaphor *V_S* generiert, das mit 1 initialisiert wird. Für jede Anweisung

    region *v* do S

wird vom Compiler folgender Code erzeugt:

    P(*V_S*)
      S
    V(*V_S*)

Es ist klar, daß dadurch wechselseitiger Ausschluß gewährleistet ist.

Kritische Bereiche können auch geschachtelt auftreten. Dabei muß man sich allerdings der Gefahr eines Deadlocks in solchen Konstruktionen bewußt sein, wie das folgende Beispiel zeigt.

Beispiel 2.3

```
var v,w : T
   parbegin
      Q: region v do region w do S1
      R: region w do region v do S2
   parend
```

Es ist nun durchaus möglich, daß Q und R gleichzeitig in ihre kritischen Bereiche *v* und *w* eintreten.

- Versucht Q in seinen Bereich *w* einzutreten, wird er, weil R bereits innerhalb seines Bereichs *w* ist, blockiert.
- Versucht R in seinen Bereich *v* einzutreten, wird er, weil Q bereits innerhalb seines Bereiches *v* ist, blockiert.

Damit ist zwischen Prozeß Q und R eine Deadlock-Situation entstanden. Die Deadlock-Problematik wird später noch ausführlicher behandelt werden.

Bedingte kritische Bereiche

Eine Erweiterung der kritischen Bereiche stellt das Synchronisationsmittel der **bedingten kritischen Bereiche** dar. Ein Prozeß darf seine kritische Sektion erst dann ausführen, wenn eine bestimmte Bedingung erfüllt ist. Diese Bedingung ist dem *Wachkommando* (guard command) von Dijkstra [Dijk,75] sehr ähnlich. Eine Variante des Wachkommandos ist als *selective wait* in der Programmiersprache ADA implementiert [Geh,84]. Der grundsätzliche Unterschied zum kritischen Bereich liegt in der Bereichs-Anweisung, welche folgende Syntax hat:

region *v* **when** *B* **do** *S* (B: logischer Ausdruck)

Andere Notationen verwenden statt **when** die Bezeichnung **await**. Ebenso wie bei kritischen Bereichen schließen sich bedingte kritische Bereiche, die sich auf dieselbe gemeinsame Variable beziehen, gegenseitig aus. Wenn ein Prozeß versucht seine kritische Sektion zu betreten, wird der logische Ausdruck *B* ausgewertet. Ist er wahr, wird die Anweisungsfolge *S* ausgeführt. Ist er falsch, verzichtet erstens der Prozeß auf den wechselseitigen Ausschluß, zweitens wird er verzögert, bis *B* wahr wird, und drittens befindet sich kein Prozeß in dem mit *v* assoziierten Bereich [PeSi,85]. Dieses Konzept soll nun zur Lösung des **Erzeuger/Verbraucher**-Problems mit *beschränktem Puffer* [BrHa,77a] angewendet werden.

Beispiel 2.4

Ein Prozeß, der Erzeuger, produziert Information, die vom Verbraucher konsumiert wird. Beispielsweise produziert ein Druckertreiber Zeichen, die vom Drucker ausgedruckt werden.
Damit solche Prozesse parallel ablaufen können, muß ein **Puffer** zur Verfügung gestellt werden. Dieser besteht hier aus **n** identischen Elementen, die in einem Kreis angeordnet sind. Der Kreis enthält eine Reihe von leeren

Elementen, die vom Erzeuger gefüllt, und eine Reihe von vollen Elementen, die vom Verbraucher geleert werden können.
Darstellen läßt sich dieser Puffer mittels eines zirkulären Arrays (s. Abb. 4) mit zwei Zeigern ein und aus. Die Variable ein zeigt auf den nächsten freien Platz des Puffers, während aus auf das erste volle zu konsumierende Element zeigt.

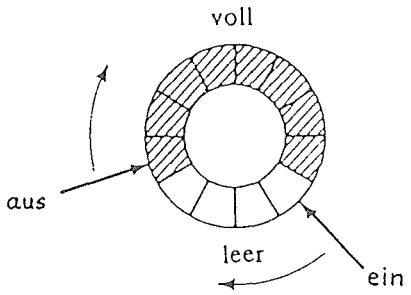


Abb. 4. Zirkuläres Array mit den Zeigern ein und aus.

Die benötigte Datenstruktur sieht folgendermaßen aus:

```
type Element = ...
     Zirk_array = record
                    Puffer : array[0..n-1] of Element
                    ein,aus : 0..n-1
                    Zähler : 0..n
                  end
var v : Zirk_array
    ep,ev : Element        {produzierte und konsumierte Information}
```

Die Variablen Zähler, ein und aus werden mit 0 initialisiert. Zähler wird nach jeder "Produktion" um 1 erhöht und nach jedem "Verbrauch" um 1 erniedrigt.

Der Erzeugerprozeß fügt ein Element ep in den Puffer ein, indem er ausführt:

```
region v when Zähler < n do
   begin
      Puffer[ein] := ep
      ein := (ein + 1) mod n
      Zähler := Zähler + 1
   end
```

Der Verbraucherprozeß entfernt ein Element ev aus dem Puffer durch:

```
region v when Zähler > 0 do
   begin
      ev := Puffer[aus]
      aus := (aus + 1) mod n
      Zähler := Zähler - 1
   end
```

Obwohl kritische Bereiche mehr Schutz vor Programmierfehlern bei der Synchronisation von Prozessen bieten als beispielsweise Semaphore, zeigt das obige Beispiel, daß jeder Prozeß seine eigene Synchronisationsanweisung benötigt. Dieser Nachteil wird durch einen anderen Synchronisationsmechanismus ausgeschaltet.

2.3.4 Monitore

Das Synchronisationskonzept, welches den oben angesprochenen Nachteil kritischer und bedingter kritischer Bereiche behebt, ist der **Monitor**.
Ein *Monitor* besteht aus einer Menge *globaler Variablen* und einigen (eventuell parametrisierten) Prozeduren, die Operationen auf diesen Variablen ausführen. Auf die globalen Variablen kann nur innerhalb des Monitors zugegriffen werden. Die Ausführung der Prozeduren in einem Monitor ist wechselseitig ausgeschlossen. Dies stellt sicher, daß auf die globalen Variablen niemals gleichzeitig zugegriffen werden kann. Monitorprozeduren

können auch lokale Variablen enthalten. Die Kommunikation mit den Prozessen geschieht über die Parameter der Prozeduren. Daneben hat ein Monitor noch einen Initialisierungsteil für die globalen Variablen, der ein einziges Mal vor den Prozeduraufrufen ausgeführt wird.
Weil der Monitor ein statisches Objekt ist (Variablen- und Prozedurdeklarationen), ergibt sich aus dem Aufruf einer Monitorprozedur die einzige Möglichkeit für einen Prozeß, einen Monitor auszuführen [Ari,82].

Ein Monitor mit Prozeduren proc1, ..., procN hat folgende allgemeine Struktur [HwBr,84]:

```
monitor Name
   var ...                      {Deklaration globaler Variablen}
   procedure proc1(parameter)
      var ...                   {Deklaration lokaler Variablen}
      begin
         {Anweisungen um proc1 zu implementieren}
      end

   ...

   procedure procN(parameter)
      var ...                   {Deklaration lokaler Variablen}
      begin
         {Anweisungen um procN zu implementieren}
      end
begin
   {Initialisierung globaler Variablen}
end.
```

Um die Synchronisation in Monitoren zu realisieren, bedient man sich einer *Bedingungsvariablen*, mit deren Hilfe man Prozesse verzögern kann. Eine solche Variable kann nur innerhalb eines Monitors deklariert werden [Ari,82]. Auf ihr sind zwei Operationen definiert: **signal** und **wait**.
Sei x eine Bedingungsvariable. Ein Prozeß, der **wait**(x) ausführt, wird in eine Warteschlange von Prozessen eingeordnet, die ebenfalls wegen **wait**(x) blockiert sind. Nach Ausführung dieser Anweisung wird der Eintritt in den Monitor wieder freigegeben. Wenn ein Prozeß **signal**(x) ausführt und die

Warteschlange von x nicht leer ist, wird der erste Prozeß der Warteschlange erweckt (arbeitet nach dem FIFO-Prinzip). Waren keine Prozesse in der Warteschlange von x, dann wirkt die Ausführung von **signal**(x) wie ein Leerbefehl.
Die Anforderung, nach Ausführung von **wait**(x) den Eintritt in den Monitor wieder freizugeben, ist offensichtlich. Wenn ein Prozeß P_1 blockiert ist und auf das Eintreten von x wartet, so muß irgendein Prozeß P_2 das Eintreten von x durch **signal(x)** anzeigen. Würde P_1 den wechselseitigen Ausschluß zum Eintritt in den Monitor nicht freigegeben haben, könnte P_2 niemals eintreten, damit **signal**(x) nicht ausführen, und es käme zu einem Deadlock. Ein Prozeß, der durch **wait**(x) blockiert wurde und später durch **signal**(x) wieder aufgeweckt wird, hat Vorrang gegenüber solchen Prozessen, die versuchen den Monitor mit einem Prozeduraufruf zu betreten. Diese Forderung nach *sofortiger Wiederaufnahme* führt dazu, daß ein "signalisierender" Prozeß P kurzfristig verzögert wird. Gibt der durch P erweckte Prozeß den Monitor wieder frei, dann hat P Priorität vor allen anderen Prozessen, die eintreten wollen.

Bisher sind einige parallele Programmiersprachen implementiert worden, die Monitore zur Synchronisation des Zugriffs auf gemeinsame Variablen verwenden. Zwei davon sind beispielsweise Concurrent PASCAL [BrHa,77b] und EDISON.

2.4 MESSAGE PASSING

Bisher wurden zur Kommunikation und Synchronisation mehrerer Prozesse gemeinsame Variablen verwendet. Ein anderes Konzept, bei dem sich die Kommunikation und Synchronisation durch den Austausch von *Nachrichten* vollzieht, ist **Message Passing**. Die Prozesse eines Systems senden und empfangen diese Nachrichten über eine **Kommunikationsverbindung** (communication channel). Diese Kommunikationsverbindung kann durch eine geschützte gemeinsame Datenstruktur implementiert sein, die nur durch das Betriebssystem verändert werden kann.

Kommunikation wird bewerkstelligt, indem einem Empfänger mit dem Empfang einer Nachricht auch bestimmte Daten von einem Senderprozeß übermittelt werden.
Synchronisation bedeutet hier, daß eine Nachricht erst dann empfangen werden kann, nachdem sie auch abgeschickt worden ist [AnSch,83].
Wollen Prozesse P und Q miteinander in Verbindung treten, stehen ihnen zwei Operationen zur Verfügung.

- **send** *Ausdruckliste* **to** *Zielbezeichner*:
  Wenn das **send** ausgeführt wird, enthält die Nachricht die Werte der Ausdrücke in *Ausdruckliste*. Der *Zielbezeichner* gibt an, wohin die Nachricht geschickt wird.

- **receive** *Variablenliste* **from** *Quellbezeichner*:
  Variablenliste ist eine Menge von Variablen. Der *Quellbezeichner* zeigt an, woher die Nachricht kommt. Durch den Empfang der Nachricht werden die Daten der Nachricht den Variablen in *Variablenliste* zugewiesen.

Diese Operationen führen zu zwei Problemen, die im folgenden behandelt werden sollen:

1. Wie können Quell- und Zielbezeichner spezifiziert werden?
2. Wie wird die Kommunikation zwischen Quelle und Ziel synchronisiert?

2.4.1 Kommunikationsarten

Eine Kommunikationsverbindung wird durch Quell- und Zielbezeichner festgelegt. Sie läßt sich am einfachsten dadurch spezifizieren, daß man die Prozeßnamen als Quelle und Ziel verwendet. Die Kommunikation über diese Art der Spezifikation nennt man *direkte Kommunikation*. Zwei Prozesse, ein Sender P und ein Empfänger Q, tauschen wie folgt Informationen aus:

P: **send** Nachricht **to** Q
Q: **receive** Nachricht **from** P

Diese Kommunikationsverbindung hat folgende Eigenschaften [PeSi,85]:

- Zwischen jedem Prozeßpaar, das kommunizieren möchte, wird automatisch eine Verbindung eingerichtet. Zur Kommunikation brauchen die Prozesse nur die Identität jedes anderen Prozesses zu kennen, mit dem sie kommunizieren möchten.
- Zwischen jedem Paar kommunizierender Prozesse gibt es genau eine Verbindung.

Dieses Schema zeigt eine Symmetrie in der Adressierung, denn beide, Sender und Empfänger, müssen einander benennen, um zu kommunizieren. In der Programmiersprache CSP (Communicating Sequential Processes, [Hoa,78]) ist diese Form der symmetrischen Kommunikation realisiert.
Eine asymmetrische Form der Kommunikation gibt es beispielsweise in den Programmiersprachen DP (Distributed Processes, [BrHa,78]) und ADA, wo der Empfänger den Namen des sendenden Prozesses weder benennt noch kennt.
Der Nachteil dieses direkten Schemas ist die begrenzte Modularität der Prozeßdefinitionen. Die Änderung eines Prozeßnamens kann die Überprüfung aller anderen Prozeßnamen nach sich ziehen. Alle Referenzen auf den alten Namen müssen gefunden und auf den neuen Namen geändert werden.

Eine andere Art der Kommunikation, die diesen Nachteil behebt, ist die *indirekte Kommunikation*. Wollen Prozesse miteinander *indirekt* kommunizieren, schicken und empfangen sie Nachrichten zu bzw. von einem *Briefkasten* (mailbox) [AnSch,83]. Ein Briefkasten tritt dabei sowohl als Zielbezeichner eines Senderprozesses als auch als Quellbezeichner eines Empfängerprozesses auf.
Sei A ein Briefkasten, P der Sender und Q der Empfänger. Die Kommunikation zwischen P und Q läuft dann folgendermaßen ab:

P: **send** Nachricht **to** A
Q: **receive** Nachricht **from** A

Die zu einem Briefkasten A geschickten Nachrichten, können von jedem Prozeß, der ein **receive** auf A ausführt, empfangen werden.

Die indirekte Kommunikation hat folgende Eigenschaften [PeSi,85]:

- Eine Verbindung zwischen zwei Prozessen wird nur dann eingerichtet, wenn sie einen gemeinsamen Briefkasten haben.

- Eine Verbindung kann mehr als zwei Prozessen zugeordnet sein.

Neben dem allgemeinen Fall, wo ein Briefkasten als Ziel- bzw. Quellbezeichner mehrerer verschiedener Sender bzw. Empfänger auftreten kann, gibt es einen Spezialfall des Briefkastenkonzepts. Dabei kann der Name eines Briefkastens nur in der **receive**-Anweisung *eines* Prozesses als Quellbezeichner vorkommen [AnSch,83]. Dieses Konzept eignet sich gut für solche Systeme, in denen einem Briefkasten mehrere Sender und ein Empfänger zugeordnet sind.

2.4.2 Synchronisation

Nach der Spezifikation der Kommunikationsverbindungen zwischen den Prozessen stellt sich nun die Frage, ob und wie die Ausführung einer Message-Passing-Anweisung eine Verzögerung bewirken kann. Viele Message Passing Konzepte verwenden zur Synchronisation einen **Verbindungspuffer**. Die Nachrichten werden zwischen Sende- und Empfangszeitpunkt gepuffert. Jeder Verbindungspuffer besitzt eine gewisse Kapazität, die festlegt, wieviele Nachrichten er zeitweise enthalten kann [AnSch,83].

1. **Nullkapazität** besagt, daß es keine Pufferung gibt, d.h. die Verbindung kann keine wartenden Nachrichten enthalten. Bei Ausführung eines **send** wird der Sender immer solange verzögert, bis ein übereinstimmendes **receive** des Empfängers ausgeführt wird (Übereinstimmung ist durch Quell- und Zielbezeichner festgelegt).
   Auf der anderen Seite bewirkt die Ausführung eines **receive** eine Verzögerung des Empfängers, wenn keine übereinstimmende Nachricht vorliegt. Wird die Nachricht durch das entsprechende **send** verfügbar, kann der Empfänger fortfahren.
   Beide Prozesse müssen also synchronisiert werden, damit ein Nachrichtentransfer stattfinden kann. Man bezeichnet dies als *synchrones Message Passing* und die Art der Synchronisation nennt man

Rendezvous. Wenn synchrones Message Passing verwendet wird, stellt ein Nachrichtenaustausch immer einen Synchronisationspunkt in der Ausführung des Senders und Empfängers dar.

2. **Beschränkte Kapazität** bedeutet, daß im Puffer eine endliche Anzahl von Nachrichten enthalten sein kann. Wenn eine neue Nachricht abgeschickt wird und der Puffer nicht voll ist, wird die Nachricht in den Puffer eingereiht, und der Sender kann ohne zu warten mit seiner Arbeit fortfahren. Ist der Puffer voll, gibt es zwei Möglichkeiten:

   a. Der Sender wird solange verzögert, bis wieder ausreichend Platz im Puffer ist.
   b. Der Sender erhält eine Meldung, die anzeigt, daß die Nachricht wegen Platzmangels im Puffer nicht gesendet werden konnte.

   Man spricht in diesem Zusammenhang von *gepuffertem Message Passing*. Ist der Puffer leer und ein Prozeß führt ein **receive** aus, gibt es ebenfalls zwei Möglichkeiten:

   a. Der Empfänger wird solange verzögert, bis wieder eine Nachricht im Puffer ist.
   b. Die Ausführung der **receive**-Anweisung endet mit einer Meldung, daß keine Nachricht vorhanden war.

3. Bei **unbeschränkter Kapazität** besitzt der Puffer eine potentiell unendliche Länge, d.h. eine unbeschränkte Anzahl von Nachrichten kann in ihm warten. Der Sender wird niemals verzögert (der Empfänger nur dann, wenn der Puffer leer ist). Da vom Sender ausgehend keine Synchronisation nötig ist, nennt man diese Art des Nachrichtentransfers *asynchrones Message Passing* oder *send no-wait*.

Für die Verzögerung der Prozesse ist das Betriebssytem verantwortlich, das einen wartenden Prozeß in eine Warteschlange einreiht und ihn bei Ausführung einer übereinstimmenden Message-Passing-Anweisung wieder aktiviert. Eine *blockierende* Form der **receive**-Anweisung ist auch die gebräuchlichere, weil ein Empfänger meist nichts anderes zu tun hat, als auf den Empfang der Nachricht zu warten.

Eine nichtblockierende Form der **receive**-Anweisung erlaubt einen Test darauf, ob ihre Ausführung zu einer Blockierung führen würde. Dadurch kann ein Prozeß alle verfügbaren Nachrichten prüfen und eine zum Weitermachen auswählen. Beispielsweise erlaubt die Ausführung der **receive**-Variante

receive *Variablenliste* **from** *Quellbezeichner* **when** B

nur den Empfang derjenigen Nachrichten, die die Bedingung B erfüllen.

Eine der Programmiersprachen, in die das Konzept des Message Passing integriert wurde, ist CSP. CSP bedient sich des synchronen Message Passing und der direkten Kommunikation. Zur Kommunikation und Synchronisation werden sogenannte *Ein-* und *Ausgabekommandos* verwendet.

2.5 DAS DEADLOCK-PROBLEM

Während der parallelen Ausführung von Programmen konkurrieren mehrere Prozesse um eine endliche Anzahl von Betriebsmitteln. Die Betriebsmittel eines Rechnersystems können nach verschiedenen Kriterien unterteilt werden [KNU,75]. Zur Behandlung des Deadlock-Problems lassen sich folgende Unterscheidungen treffen:

- **Entziehbare Betriebsmittel** sind Betriebsmittel, die einem Prozeß von außen entzogen und einem anderen Prozeß zugeteilt werden können. Zum Zeitpunkt des Entzugs kann der Betriebsmittelzustand gerettet und später wiederhergestellt werden. Ein Beispiel ist der Hauptspeicher, dessen Inhalt beim Entzug auf einem externen Speicher sichergestellt werden muß. Die Entzugsstrategie wird i. allg. durch die entstehenden Kosten des Entzugs beeinflußt.

- **Nichtentziehbare Betriebsmittel** sind Betriebsmittel, die von dem Prozeß, der sie benutzt hat, explizit freigegeben werden müssen. Sie unterscheiden sich von den entziehbaren Betriebsmitteln meist nur durch die Höhe der Kosten, die bei einem Entzug anfallen würden. Beispiels-

weise wäre der Entzug einer Magnetbandeinheit unter Umständen zu teuer.

- **Exklusiv verwendbare Betriebsmittel** sind Betriebsmittel, die nur von einem Prozeß gleichzeitig verwendet werden können. Ein exklusives Betriebsmittel ist beispielsweise ein Drucker.

- Bei **nichtexklusiv verwendbaren Betriebsmitteln** können mehrere Prozesse dasselbe Betriebsmittel gleichzeitig verwenden. Die in ihm enthaltenen Informationen dürfen sich durch die gleichzeitige Benutzung nicht ändern. Ein Beispiel ist eine Datei, aus der mehrere Prozesse gleichzeitig lesen können.

Alle globalen (für alle Prozesse verfügbaren) Betriebsmittel B seien in m Betriebsmittelklassen R_k, $1 \leq k \leq m$, unterteilt, wobei die Betriebsmittel einer Klasse ihren Eigenschaften zufolge äquivalent sind. Für die Betriebsmittelklassen gilt dann:

$$B = R_1 \cup R_2 \cup \ldots \cup R_m \text{ und}$$
$$R_i \cap R_j = \emptyset, \text{ für alle } i \neq j,\ 1 \leq i,j \leq m.$$

Jeder Prozeß des Systems beansprucht ein oder mehrere Betriebsmittel. Falls diese zu gegebener Zeit nicht verfügbar sind, kommt der Prozeß in einen Wartezustand, er wird blockiert. Es kann nun passieren, daß blockierte Prozesse ihren Zustand nie mehr ändern, weil die von ihnen angeforderten Betriebsmittel von anderen, ebenfalls auf bestimmte Betriebsmittel wartenden Prozessen festgehalten werden. Diese Situation nennt man **Deadlock**.

Eine Deadlock-Situation besteht beispielsweise in einem System mit mehreren Prozessen, die unendlich lange vom Eintritt in ihre kritische Sektion ausgeschlossen bleiben. Bei geschachtelten kritischen Bereichen in Beispiel 2.3 konnte die Situation eintreten, wo jeder der beiden Prozesse ein nichtentziehbares, exklusives Betriebsmittel (kritischen Bereich) besitzt, das vom jeweils anderen Prozeß benötigt wird. Dadurch sind beide Prozesse blockiert und es kommt zu einem Deadlock. Wurde ein Prozeß bei Benutzung eines Monitors blockiert, mußte er zur Vermeidung einer Deadlock-Situation den Eintritt in das exklusive Betriebsmittel Monitor freigeben.

Definition 2.5

Eine Menge von Prozessen befindet sich im *Deadlock*-Zustand, wenn jeder der Prozesse in dieser Menge auf ein angefordertes Betriebsmittel wartet, das nur von Prozessen aus dieser Menge freigegeben werden kann.

Eine Deadlock-Situation kann nur dann eintreten, wenn die folgenden vier Bedingungen im System bestehen [Weck,82]:

1. **Wechselseitiger Ausschluß** ("mutual exclusion"): Die betreffenden Prozesse benötigen exklusiven Zugriff auf die jeweiligen Betriebsmittel.

2. **Warten auf Betriebsmittel** ("wait for"): Die Prozesse verfügen schon über ihnen zugeteilte Betriebsmittel, während sie auf weitere Betriebsmittel warten.

3. **Nichtentziehbarkeit** ("nonpreemption"): Es ist nicht möglich, Prozessen einmal zugeteilte Betriebsmittel zu entziehen, wenn die Benutzung dieser Betriebsmittel nicht abgeschlossen worden ist.

4. **Zirkuläres Warten** ("circular wait"): In einem System von n Prozessen gibt es eine geschlossene Kette von blockierten Prozessen $\{P_1, P_2, \ldots, P_i\}$, $1 < i \leq n$, in der jeder der Prozesse über gewisse Betriebsmittel verfügt, die vom nächsten Prozeß in der Kette benötigt werden.

Wenn die ersten drei dieser Bedingungen eintreten, dann ist die Entstehung von Deadlocks im Prinzip möglich; eine Deadlock-Situation muß jedoch nicht eintreten, solange verhindert werden kann, daß sich eine geschlossene Kette von aufeinander wartenden Prozessen bildet. Umgeht man auf diese Art Deadlock-Situationen, so spricht man von *Vermeidung* (avoidance) von Deadlocks. Wendet man dagegen eine Strategie an, die durch Negation der oben genannten Bedingungen das Entstehen von Deadlocks von vornherein ausschließt, so bezeichnet man dies als *Verhinderung* (prevention) von Deadlocks.
In einem System, in dem sich Deadlocks weder verhindern noch vermeiden lassen, kann das Deadlock-Problem doch gelöst werden, wenn man Prozessen, die an einem Deadlock beteiligt sind, Betriebsmittel gewaltsam entziehen

kann. In diesem Falle stellt sich die Aufgabe Deadlock-Situationen zu *entdecken* (detection) und zu *beseitigen* (recovery). Bei solchen Verfahren spielt die Höhe der anfallenden Kosten, die durch den Entzug entstehen, eine wichtige Rolle. Wie schon erwähnt, können diese Kosten je nach Betriebsmittelart unterschiedlich hoch sein.

2.5.1 Vermeidung von Deadlocks

Sollen in einem System, in dem die für das Eintreten von Deadlocks notwendigen Bedingungen bestehen können, Deadlocks vermieden werden, braucht man zusätzliche Information darüber, wieviele Betriebsmittel jeder Prozeß des Systems benötigen wird. Verfügt man über solche Vorweginformationen, dann lassen sich Algorithmen konstruieren, die sicherstellen, daß ein System nie in einen Deadlock-Zustand gelangt.
Ein Algorithmus zur Vermeidung von Deadlocks unternimmt eine dynamische Überprüfung des Betriebsmittelvergabezustandes, um die Entstehung einer zirkulären Wartebedingung zu vermeiden.
Der **Zustand** für die Betriebsmittelvergabe ist definiert durch die Anzahl verfügbarer und zugeteilter Betriebsmittel sowie durch die Zahl der maximalen Anforderungen von Betriebsmitteln der Prozesse. Ein System befindet sich in einem **sicheren Zustand**, wenn es eine **sichere Folge** gibt.

Definition 2.6

Gegeben sei eine Folge von Prozessen $<P_1, P_2, \ldots, P_n>$, die bez. ihrer Prozeßnummern linear geordnet sind. Eine solche Folge von Prozessen ist eine *sichere Folge*, wenn für jeden Prozeß P_i, $i = 1, 2, \ldots, n$, die Betriebsmittelanforderungen durch die verfügbaren und durch die an alle anderen Prozesse P_j, mit $j < i$, gebundenen Betriebsmittel erfüllt werden können.

Ein System befindet sich also in einem *sicheren Zustand*, wenn es einen Weg gibt, allen Prozessen die von ihnen angeforderten Betriebsmittel so zuzuteilen, daß jeder Prozeß seine Arbeit beenden kann. Gibt es keinen solchen Weg, wird der Zustand des Systems als gefährdet bezeichnet.

Aus Definition 2.6 ergeben sich folgende Konsequenzen [PeSi,85]:

- Deadlock-Zustände sind gefährdete Zustände.
- Befindet ein System sich in einem gefährdeten Zustand, impliziert dies keinen Deadlock-Zustand; vielmehr ist dies eine Situation, die zu einem Deadlock-Zustand führen *kann*.
- Fordert ein Prozeß ein Betriebsmittel an, welches gerade verfügbar ist, muß er eventuell trotzdem noch warten, weil möglicherweise noch nicht alle von ihm benötigten Betriebsmittel verfügbar sind. Dadurch wird unter Umständen die Ausnutzung der Betriebsmittel verschlechtert.

Mit dem Konzept des sicheren Zustandes können Deadlock-Vermeidungsalgorithmen entworfen werden, die sicherstellen, daß ein System nie in einen gefährdeten oder einen Deadlock-Zustand kommt. Wenn ein Prozeß ein verfügbares Betriebsmittel anfordert, muß das System entscheiden, ob das Betriebsmittel sofort zugeteilt werden kann oder ob der Prozeß warten muß. Die Anforderung wird nur dann bewilligt, wenn bei ihrer Zuteilung das System in einem sicheren Zustand bleibt [HwBr,84].

2.5.2 Verhinderung von Deadlocks

In einem System, in dem die Entstehung von Deadlocks a priori ausgeschlossen sein soll, muß jederzeit mindestens eine der vier Bedingungen negiert sein.
Da es i. allg. nicht möglich ist, Deadlocks durch Negation der ersten Bedingung zu verhindern (es gibt Betriebsmittel, die exklusiv sein müssen), lassen sich durch Negation der restlichen drei Bedingungen drei grundlegende Strategien zur Verhinderung von Deadlocks festlegen [Weck,82].

- **Kein Warten auf Betriebsmittel** ("no wait for"): Jeder Prozeß muß seinen Bedarf an Betriebsmitteln auf einmal verlangen und darf erst bei vollständiger Erfüllung seiner Anforderungen fortfahren.
  Dies kann zu schlechter Ausnutzung der Betriebsmittel führen, weil einige davon unter Umständen für einen langen Zeitraum unbenutzt bleiben müssen.

- **Entziehbarkeit** ("preemption"): Wird einem Prozeß der Zugriff auf ein Betriebsmittel versagt, muß er alle in seinem Besitz befindlichen Betriebsmittel freigeben und eine neue Gesamtanforderung stellen.
  Diese Strategie ist nur auf solche Betriebsmittel anwendbar, die einem Prozeß entzogen und nachher wieder zugeteilt werden können (z.B. ein Prozessor). Der Zustand der zu entziehenden Betriebsmittel muß gespeichert und später wiederhergestellt werden.

- **Kein zirkuläres Warten** ("no circular wait"): Die Betriebsmittelklassen $R_1, \ldots, R_m$ unterliegen einer linearen Ordnung, $R_1 < R_2 < \ldots < R_m$. Jeder Betriebsmittelklasse ist eine eindeutige ganze Zahl zugeordnet. Verfügt ein Prozeß über ein Betriebsmittel der Klasse R_i, dann kann er ein Betriebsmittel der Klasse R_j nur dann anfordern, wenn $R_j > R_i$ ist. Durch diese Regel wird die Bildung einer geschlossenen Kette von blockierten Prozessen verhindert.

2.5.3 Entdeckung und Beseitigung von Deadlocks

Werden in einem System keine Vorkehrungen getroffen, um die Möglichkeit der Entstehung von Deadlocks a priori auszuschließen, ist es unumgänglich Entdeckungs- und Beseitigungsschemata für Deadlocks zu implementieren.
Ein Algorithmus zur Entdeckung von Deadlocks wird periodisch aufgerufen, um festzustellen, ob ein Deadlock eingetreten ist. Um diese Feststellung treffen zu können, benötigt er ausführliche Kenntnis über den Betriebsmittelzustand des Systems. Die zu einem bestimmten Zeitpunkt benötigten Informationen könnten in den folgenden Datenstrukturen festgehalten sein [PeSi,85].

Verfügbar: ein Vektor der Länge *m*. Verfügbar[j] = k heißt, daß k Betriebsmittel der Betriebsmittelklasse j verfügbar sind.

Zugeteilt: eine $n \times m$ Matrix. Zugeteilt[i,j] = k besagt, daß Prozeß P_i momentan k Betriebsmittel der Betriebsmittelklasse j zugeteilt sind.

Angefordert: eine $n \times m$ Matrix. Ist Angefordert[i,j] = k, dann werden k Betriebsmittel der Betriebsmittelklasse j von Prozeß P_i angefordert.

Ein Algorithmus, der aufgrund dieser Informationen Deadlocks entdeckt, kann [CES,71] entnommen werden. Er untersucht alle möglichen Zuteilungsfolgen der Prozesse, die noch zu beenden sind.

Hat ein Entdeckungsalgorithmus eine eingetretene Deadlock-Situation ermittelt, kann man diese nur dadurch beheben, indem man entweder

1. einen oder mehrere der beteiligten Prozesse gewaltsam abbricht ("crash") und seine Betriebsmittel freigibt, oder
2. einem oder mehreren der beteiligten Prozesse zwangsweise so viele Betriebsmittel entzieht, daß der Deadlock aufgehoben wird ("forced preemption").

Das erste dieser Verfahren ist so drastisch, daß es möglicherweise zu einem völligen Neustart des Systems führen kann.
Bei der zweiten Methode ist zu berücksichtigen, daß der Entzug der Betriebsmittel mit bestimmten Kosten für das Retten des Status und die spätere Wiederherstellung des ursprünglichen Systemzustandes verbunden ist [CES,71]. Man ordnet beispielsweise jeder Betriebsmittelklasse j einen fixen Kostenfaktor k_j, $j = 1, 2, \ldots, m$, für den zwangsweisen Entzug eines Betriebsmittels aus R_j zu. Die einem Deadlock-Prozeß P_i zu entziehenden Betriebsmittel führen dann mit diesem Kostenfaktor zu folgenden Gesamtkosten K_i (für die zugeteilten und verfügbaren Betriebsmittel gelten dabei die oben eingeführten Bezeichnungen):

$$K_i = \sum_{j=1}^{m} k_j * g(\text{Zugeteilt}[i,j]-\text{Verfügbar}[j]) \qquad i = 1, 2, \ldots, n$$

$$\text{mit} \quad g(x) = \begin{cases} x & \text{für } x > 0 \\ 0 & \text{für } x \leq 0 \end{cases}$$

Mit Hilfe von Optimierungsstrategien läßt sich dann eine Teilmenge von Prozessen $P_1, \ldots, P_k$, $k < n$, ermitteln, so daß die Gesamtkosten für den Entzug der ihnen zugeteilten Betriebsmittel minimal werden.

Bei den Verfahren zur Beseitigung von Deadlocks kann es allerdings vorkommen, daß immer derselbe Prozeß abgebrochen wird und er dadurch seine Arbeit nie beenden kann. Diese Situation nennt man "Starvation". Es muß deshalb sichergestellt werden, daß ein Prozeß nur wenige Male abgebrochen werden kann. Erreicht wird dies, indem man die Anzahl der Abbrüche eines Prozesses in den Kostenfaktor für Abbruch und Entzug seiner Betriebsmittel mit einfließen läßt.

3.0 DAS KONZEPT DES MULTITASKING

Die Notwendigkeit einer Leistungsverbesserung hat die Entwickler von Rechnersystemen veranlaßt, neue Architekturen so zu konzipieren, daß mit ihrer Hilfe Parallelverarbeitung effektiv betrieben werden kann. Erste kommerzielle Erfolge wurden auf diesem Gebiet mit Hilfe der Vektorverarbeitung erzielt. Das Konzept des **Multitasking** hat das Ziel, unter Ausnutzung der Hardware-Fähigkeiten und entsprechender Software eines Rechnersystems die Ausführungszeit von Programmen zu verringern.
Multitasking ist eine Betriebsform, die ebenso wie das Multiprocessing die parallele Verarbeitung mehrerer Prozesse, hier **Tasks** genannt, auf mehreren Prozessoren unter Ausnutzung eines gemeinsamen Speichers bezeichnet. Wichtigstes Hauptziel des Multitasking ist die Beschleunigung von Algorithmen. Dazu ist es wichtig zu wissen, wie und wann Multitasking eingesetzt werden sollte und was mit seiner Hilfe erreicht werden kann.
Durch Multitasking wird keineswegs die zur Ausführung eines Programms notwendige Arbeit (Anzahl der Operationen) reduziert. Vielmehr hat Multitasking einen gewissen Aufwand zur Folge, der zur Erhöhung der geleisteten Arbeit führt. Dieser Aufwand entsteht durch die Initiierung und die Interaktionen paralleler Tasks und wird als **Overhead** bezeichnet. Deshalb ist es notwendig, die Multitasking-Fähigkeiten mit dem Ziel eines minimalen Overheads in einem Programm zu verwenden. Dies kann beispielsweise dadurch erreicht werden, daß man den **Parallelismus** eines Algorithmus auf der höchstmöglichen Ebene ausnutzt.
Parallelismus läßt sich auf fünf verschiedenen Ebenen ausnutzen [CRAY,222]:

1. Job-Ebene
2. Jobschritt-Ebene
3. Unterprogramm-Ebene
4. Anweisungs-Ebene
5. Elementaroperationen-Ebene

Auf der höchsten Ebene, der Job-Ebene, wird Parallelverarbeitung zwischen verschiedenen Jobs betrieben. Dabei können mehrere, voneinander unabhängige Jobs auf verschiedenen Prozessoren parallel ausgeführt werden. Auf

der Jobschritt-Ebene ist die parallele Ausführung unabhängiger Teile desselben Jobs möglich. Auf der dritten Ebene ist die parallele Ausführung von Unterprogrammen oder Programmteilen desselben Programms vorgesehen. Die Multitasking-Fähigkeiten der Rechner der Firma CRAY RESEARCH bedienen sich des Parallelismus von Programmen auf dieser Ebene. Auf der Anweisungs-Ebene wird Parallelismus zwischen verschiedenen Anweisungen ausgenutzt. Dieser Ebene bedient sich auch die Vektorverarbeitung durch parallele Ausführung skalarer Operationen in DO-Schleifen. Die niedrigste Ebene bezieht sich auf die parallele Ausführung von Elementaroperationen. Die Ebene der Elementaroperationen ist meist direkt durch Hardware-Mittel implementiert. Aus dieser Gliederung wird deutlich, daß die Komplexität eines Prozesses beim Übergang von der niedrigen zur höheren Ebene zunimmt.

3.1 SPEEDUP DURCH MULTITASKING

Multitasking verringert, wie schon erwähnt, die Ausführungszeit (*turnaround*-Zeit) eines Programms, wenn die Multitasking-Fähigkeiten in geeigneter Weise ausgenutzt werden. Mit Ausführungszeit sei im folgenden die Zeit vom Start eines Programms bis zu dessen Ende gemeint. Die Beschleunigung (**Speedup**) eines Algorithmus hängt von seiner Eignung als paralleles Programm ab. Eignung heißt, daß die Häufigkeit der Interaktionen parallel auszuführender Teile der Komplexität des Algorithmus angemessen sein sollte. Die größte Verbesserung der Ausführungszeit eines Programms ist nur dann zu erwarten, wenn die Einflußfaktoren des Speedups in geeigneter Weise in die Ausführungszeit des parallelen Algorithmus einfließen.

3.1.1 Theoretischer Speedup und Amdahls Gesetz

Das **Gesetz von Amdahl** beruht auf der Tatsache, daß nicht immer alle Teile eines Programms parallel ausgeführt werden können. Es stellt deshalb eine Beziehung zwischen dem **theoretischen** Speedup und dem Multitasking-fähigen

Teil eines Programms her. Amdahls Gesetz geht von der Annahme aus, daß während der Ausführung eines Programms kein Overhead und keinerlei Verzögerungen auftreten [Lar,84a].

Sei T_1 die Ausführungszeit eines Programms ohne Multitasking. Durch Nutzung der Multitasking-Fähigkeiten kann ein dynamischer Teil dieses Programms parallel ausgeführt werden. Die zur sequentiellen Ausführung dieses Teils benötigte Zeit gibt, prozentual auf die Zeit T_1 bezogen, den **Multitasking-Teil f** wieder. Die **theoretische Ausführungszeit** T_{th} (ohne Overhead und ohne Verzögerungen) setzt sich dann aus folgenden Komponenten zusammen [Lar,84a]:

$T_{th} = T_s + T_m$ mit
$T_s = (1-f) * T_1$: Zeit für sequentiellen Teil
$T_m = (f/p) * T_1$: Zeit für Multitasking-Teil
(p = Anzahl Prozessoren)

Daraus ergibt sich als theoretisch erreichbarer Speedup in Abhängigkeit von f und p das *Gesetz von Amdahl*:

$$S_{th}(p,f) = \frac{T_1}{T_s + T_m} = \frac{T_1}{T_1*((1-f) + (f/p))} = \frac{1}{(1-f) + (f/p)}$$

Aus Abb. 5 erkennt man, daß für eine feste Anzahl von Prozessoren der Speedup dann am höchsten ist, wenn der im Multitasking-Modus ausführbare Teil eines Programms maximal ausgeschöpft wird.

Aus Amdahls Gesetz läßt sich ein weiterer wichtiger Sachverhalt entnehmen. Abb. 6 zeigt, daß für einen festen, mit Multitasking ausführbaren Teil eines Programms ein Anstieg der Prozessorzahl nicht zu einem entsprechend großen Anstieg des Speedups führt. Der Grund dafür liegt in der Ausführungszeit für den sequentiellen Teil. Für eine hohe Anzahl von Prozessoren wird die Ausführungszeit eines Programms von dem zwingend sequentiell auszuführenden Teil dominiert, d.h. eine hohe Anzahl von Prozessoren erscheint nur dann sinnvoll, wenn nahezu das gesamte auszuführende Programm Multitasking-fähig ist.

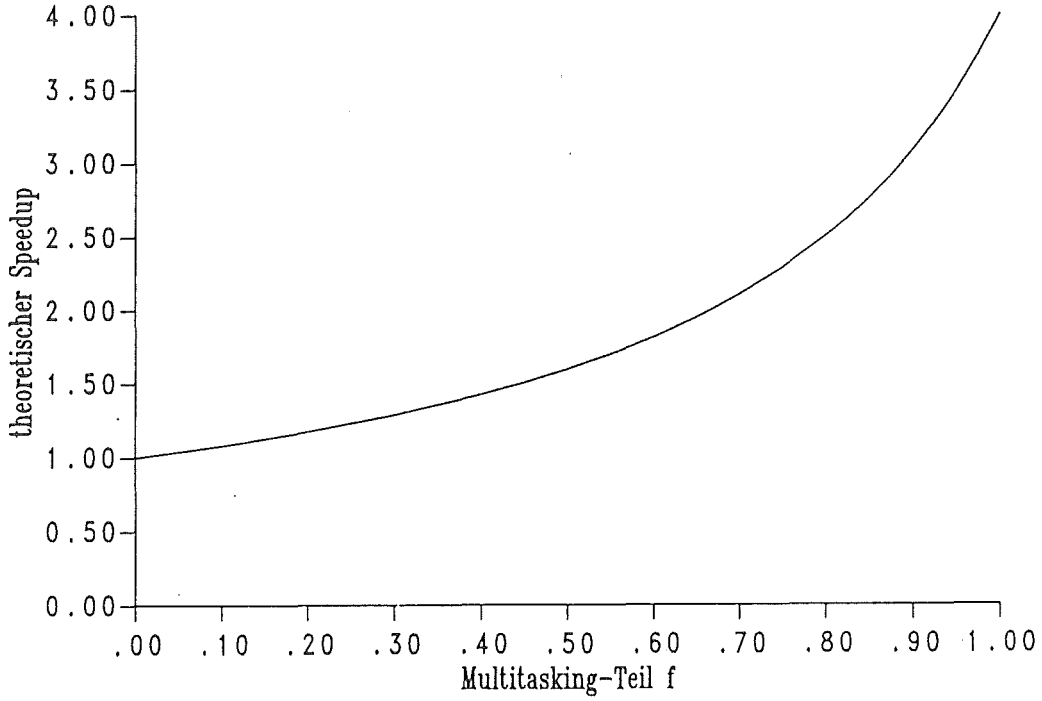


Abb. 5. Theoretischer Speedup $S_{th}(f)$ (p = 4).

In der Praxis ist ein Speedup von p jedoch nicht möglich, weil weitere Faktoren, die in Amdahls Gesetz keine Berücksichtigung finden, die Beschleunigung eines Programms begrenzen:

- Die Synchronisation paralleler Tasks bewirkt eine Verzögerung in der Ausführung der Tasks. Während dieser Verzögerung tragen diese Tasks nicht zur parallelen Ausführung bei.

- Die Ausführung eines Programms mit Multitasking hat einen gewissen Overhead zur Folge. Dieser Overhead ist umso größer, je öfter die Multitasking-Fähigkeiten ausgenutzt werden.

Bezieht man diesen Overhead in die Überlegungen für den Speedup mit ein, so ergibt sich daraus der **tatsächlich erreichbare** Speedup. Der durch die Ausführung eines Multitasking-Programms entstandene Overhead OH schlägt sich in der Zeit nieder, den Multitasking-Teil dieses Programms auszuführen.

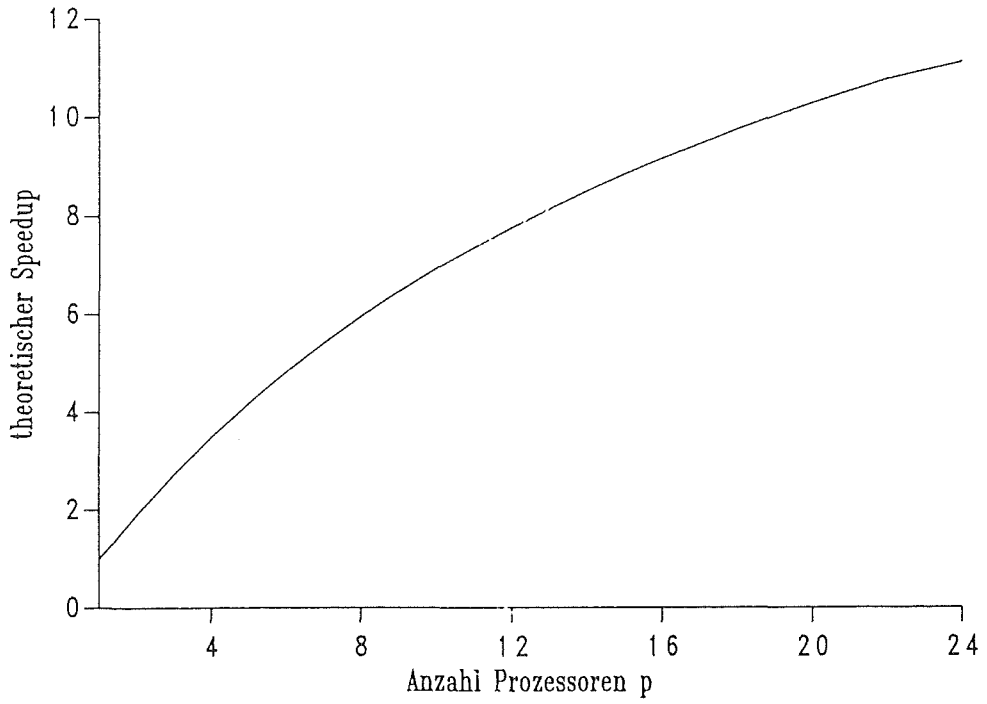


Abb. 6. Theoretischer Speedup $S_{th}(p)$ (f = 0.95).

Damit beträgt die **tatsächliche Ausführungszeit**

$$T_p = (1-f)*T_1 + (f/p)*T_1 + OH.$$

Diese Erhöhung hat folgenden tatsächlich erreichbaren Speedup (im weiteren nur noch mit Sp bezeichnet) zur Folge [NeRi,85]:

$$Sp = \frac{T_1}{T_p} = \frac{T_1}{T_1*((1-f) + (f/p)) + OH}$$

Für die Ermittlung des Speedup wird man i. allg. lediglich die sequentielle und parallele Ausführungszeit messen und das Verhältnis dieser beiden Zeiten bilden.

Den praktisch bestmöglichen Speedup erreicht man dann mit Multitasking, wenn diese Betriebsform auf **balancierte** Tasks mit ausreichend großer **Granularität** angewendet wird.

3.1.2 Granularität

Der aus der Verwendung der Multitasking-Fähigkeiten resultierende Overhead stellt eine untere Schranke für die Granularität einer Task dar. Für die Beschleunigung eines Algorithmus heißt das, daß der durch Multitasking erreichbare Zeitgewinn den bei der parallelen Ausführung entstehenden Overhead überwiegen sollte.
Die *Granularität* einer Task ist die Zeit, die benötigt wird, um einen Multitasking-fähigen Teil des ursprünglichen Programms auf einer CPU auszuführen [Lar,84a].
Der Einfluß der Granularität auf den Speedup kann wie folgt modelliert werden: Sei $X = T_g$ die Granularität eines Programmsegments. Dann beträgt die tatsächliche Ausführungszeit desselben Programmsegments auf p Prozessoren

$$T_{pg} = X*((1-f_x) + (f_x/p)) + OH.$$

Der tatsächlich erreichbare Speedup des mit Multitasking ausgeführten Programmsegments gegenüber dem ursprünglichen Programmsegment sieht dann wie folgt aus:

$$Sp = \frac{X}{X*((1-f_x) + (f_x/p)) + OH}$$

Abb. 7 zeigt den Einfluß der Granularität auf den Speedup für einen festen Overhead und einen festen Multitasking-Teil.

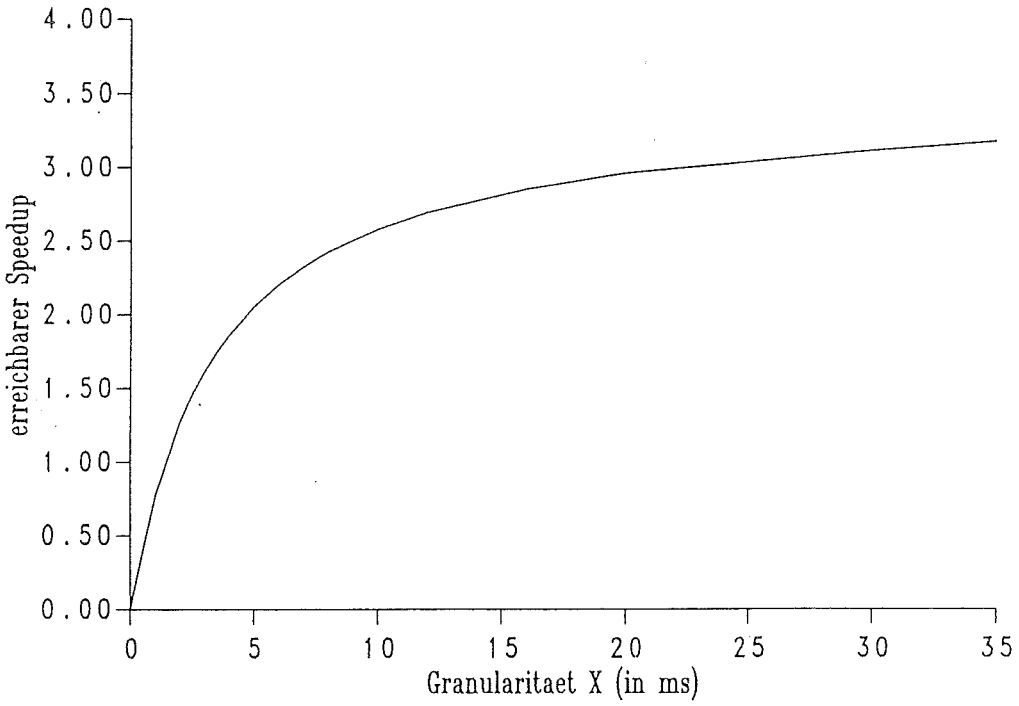


Abb. 7. Speedup als Funktion der Granularität (OH = 1 ms, f_x = 0.95, p = 4).

Durch Auflösen der obigen Formel nach der Granularität X erhält man folgende Beziehung für die **minimal** erforderliche Granularität eines Programmsegments, um bei Vorhandensein eines bestimmten Overheads und eines konstanten Multitasking-Teils einen Speedup Sp > 1 zu erzielen:

$$\frac{X}{X*((1-f_x) + (f_x/p)) + OH} > 1 \quad <==> \quad X > \frac{OH}{f_x - f_x/p}$$

Also kann für ein bestimmtes Programmsegment bei einem gegebenen Overhead OH und einem festen Multitasking-Teil f_x ein Speedup Sp auf p Prozessoren nur dann erreicht werden, wenn die Granularität des Segments mindestens X Zeiteinheiten beträgt. Ist beispielsweise OH = 1 ms und das Programmsegment zu 100% parallel ausführbar (f_x = 1), dann wird ein Speedup größer 1 auf p = 2 Prozessoren nur erreicht, wenn die Granularität größer als 2 ms ist.

Mit der Granularität eng verbunden ist eine gleichmäßige Verteilung der parallelen Anteile eines Programms auf die Prozessoren eines Systems.

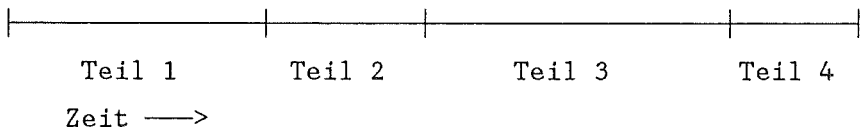
3.1.3 Balancierte Verteilung paralleler Programmteile

Die *balancierte Verteilung* ist eine Technik, die sicherstellen soll, daß der von jedem Prozessor verarbeitete Teil eines Programms annähernd gleich ist. Ziel ist es, die gesamte Arbeit, die parallel ausgeführt werden kann, möglichst gleichmäßig auf die einzelnen Tasks zu verteilen [Lar,84a].

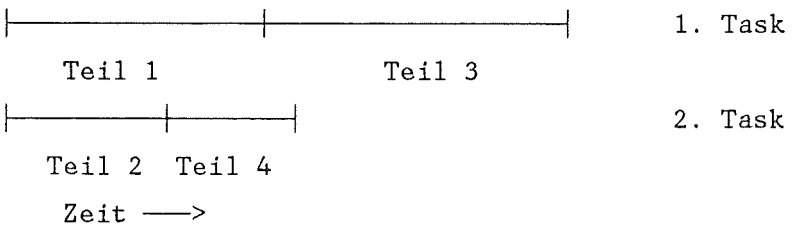
Beispiel 3.1

Balancierte Verteilung der Arbeit eines Programms auf zwei Tasks.

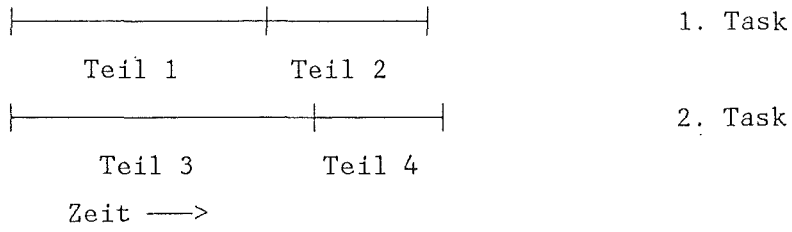
Gegeben sei ein Programm, dessen sequentieller Code sich in vier Teile aufteilen läßt:



Faßt man die Teile 1 und 3 zu einer Task sowie 2 und 4 zu einer anderen Task zusammen, so ergibt sich eine ungünstig balancierte Verteilung der Programmteile:



Damit beide Tasks annähernd die gleiche Arbeit zu leisten haben, ist es günstiger, die gesamte Arbeit wie folgt aufzuteilen:



Die balancierte Verteilung kann auf zwei Arten durchgeführt werden: **statisch** und **dynamisch**.
Die *statisch* balancierte Verteilung wird dann angewendet, wenn der Umfang an Arbeit aller Teile eines Programms schon vor deren Ausführung bestimmt werden kann. Dadurch ist eine gleichmäßige Unterteilung des Programms möglich, so daß die parallelen Tasks für ihre Arbeit dann annähernd die gleiche Zeit benötigen.

Beispiel 3.2

Statisch balancierte Verteilung der Arbeit einer DO-Schleife.

Es wird angenommen, daß für jede der N Iterationen einer DO-Schleife die Ausführungszeit annähernd gleich ist. Eine statisch balancierte Verteilung teilt jeder Task eine feste Teilmenge der N Iterationen zu. Für diese Zuteilung gibt es zwei Strategien.

1. Jeder Task wird eine Teilmenge benachbarter Iterationen zugeteilt. Der i-te Prozessor, i = 0, ..., p-1, führt dann die Iterationen (i*N) **div** p + 1 bis ((i+1)*N) **div** p aus. Für p = 2 sieht die Zuteilung wie folgt aus:

   Prozessor 0 : (0*N) **div** 2 + 1 bis ((0+1)*N) **div** 2
   Prozessor 1 : (1*N) **div** 2 + 1 bis ((1+1)*N) **div** 2

2. Die Iterationen werden den Tasks nicht zusammenhängend zugeteilt, sondern über ein konstantes Inkrement (p = 2) gesteuert:

   Prozessor 0 : $(0+k)p - 1$, $k = 1, \ldots, \lceil N/2 \rceil$
   Prozessor 1 : $(1+k)p - 2$, $k = 1, \ldots, \lfloor N/2 \rfloor$

Die *dynamisch* balancierte Verteilung kann dann zur Anwendung kommen, wenn für die Teile eines Programms der Umfang an Arbeit vor deren Ausführung noch nicht bekannt ist. Weil über diesen Arbeitsumfang nichts vorhergesagt werden kann, müssen die Tasks so konstruiert werden, daß sie dynamisch Arbeit zugeteilt bekommen und diese dann ausführen.

Beispiel 3.3

Dynamisch balancierte Verteilung der Arbeit einer DO-Schleife.

Wenn der Arbeitsumfang für die N Iterationen einer DO-Schleife größere Unterschiede aufweist, dann kann mit dynamischer Verteilung eine balancierte Aufteilung der Arbeit auf jede Task erreicht werden. Man benötigt dazu einen gemeinsamen Zähler, der die nächste auszuführende Iteration anzeigt. Jede Task benutzt und aktualisiert diesen Zähler, um dann eine oder mehrere Iterationen auszuführen. Die Benutzung des Zählers muß wechselseitig ausgeschlossen sein, d.h. die auf ihm ausgeführten Operationen müssen in eine kritische Sektion eingebettet werden. Dieser exklusive Zugriff auf den Zähler hat jedoch einen gewissen Overhead zur Folge. Ist die durchschnittliche Granularität der Iterationen groß im Vergleich zu diesem Overhead, sollten die Iterationen einzeln auf jede Task verteilt werden. Ist sie kleiner als der Overhead, sollten mehrere Iterationen auf jede Task verteilt werden.
Das folgende Programmstück zeigt eine dynamisch balancierte Verteilung der N Iterationen für große oder kleine Granularität. Sei K eine ganzzahlige, gemeinsame Variable, die die Anzahl der Iterationen angibt, die auf jede Task zu verteilen sind. Wird K = 1 gewählt, dann ist die Granularität der Iterationen groß. K > 1 bedeutet eine kleine Granularität und es werden K Iterationen auf eine Task verteilt. (I und N sind ganzzahlige, gemeinsame Variablen, L ist zu jeder Task lokal, Zähler

I wird mit 1 initialisiert). S ist ein binäres Semaphor und wird mit 1 initialisiert. Jede Task führt den folgenden Programmteil aus:

```
repeat
   P(S)
   L := I
   I := L + K
   V(S)
   {Berechnung der Iterationen L bis min(L+K-1,N)}
until L = N
```

Aus diesen Betrachtungen läßt sich für die durch Multitasking erreichbare Beschleunigung folgende Feststellung treffen: Ist die gesamte Arbeit eines Programms parallel ausführbar und alle Teile eines Programms werden balanciert verteilt, d.h. alle erzeugten Tasks haben annähernd die gleiche Granularität, dann kann die Ausführungszeit des mit Multitasking auf p Prozessoren ausgeführten Programms annähernd 1/p der Ausführungszeit auf einem Prozessor erreichen.

3.2 EFFIZIENZ, REDUNDANZ UND AUSLASTUNG EINES SYSTEMS

Die Verfügbarkeit mehrerer Prozessoren, die durch ein Programm anfallende Arbeit gleichzeitig zu leisten, ergibt im Gegensatz zu einem Einprozessorsystem einen Unterschied zwischen der Arbeit (CPU-Zyklen, Anzahl Operationen) und der Zeit diese Arbeit zu beenden. Obwohl mehrere CPUs bei der Ausführung eines Multitasking-Jobs mehr Arbeit leisten als ein äquivalentes Programm benötigt, das auf einer CPU ausgeführt wird, ist der zeitliche Aufwand für die Ausführung des Multitasking-Jobs geringer.
Sei T_1 die Zeit, um ein Programm auf einer CPU auszuführen, und $O_1 = T_1$ die gesamte dafür geleistete Arbeit, d.h. die Anzahl ausgeführter Operationen. T_p bezeichne die Ausführungszeit desselben Programms auf p Prozessoren.

Die **Effizienz** eines Algorithmus, bezogen auf ein System mit p Prozessoren, ist definiert als der Quotient aus tatsächlich erreichtem Speedup Sp und theoretisch maximal möglichem Speedup p [Kuck,78]:

$$E_p = \frac{Sp}{p} \quad \text{mit } 0 < E_p \leq 1.$$

Man erkennt hieraus, daß die Ausführung eines Multitasking-Programms nur dann effizient sein kann, wenn der resultierende Overhead niedrig ist.
Während der parallelen Ausführung eines Programms muß wegen der Verwendung der Multitasking-Fähigkeiten zusätzliche Arbeit geleistet werden. Die bei paralleler Ausführung insgesamt geleistete Arbeit sei mit O_p bezeichnet.
Die **Redundanz** einer Berechnung mißt nun diesen, durch ein Multitasking-Programm zusätzlich geleisteten Arbeitsanteil und ist definiert durch:

$$R_p = \frac{O_p}{O_1} \quad \text{mit } R_p \geq 1.$$

Streng genommen vergleicht man mit Hilfe der Redundanz zwei *verschiedene* Algorithmen zur Bearbeitung der *gleichen* Aufgabe [Hoß,81].
Ein der Effizienz gegenüberstehendes Maß für die Ausnutzung eines Systems ist dessen **Auslastung** (Utilization). Sie ist definiert durch

$$U_p = \frac{O_p}{p*T_p} \quad \text{mit } U_p \leq 1,$$

wobei $p*T_p$ die in der Zeit T_p auf p Prozessoren maximal ausführbare Anzahl von Operationen ist. Man erhält dadurch beispielsweise Aufschluß darüber, wie gut die Prozessoren eines Multiprozessorsystems bei dedizierter Ausführung ausgelastet waren, d.h. wie gut die zu leistende Arbeit verteilt war.

Im folgenden soll nun das Multiprozessorsystem **CRAY X-MP/22** der Firma CRAY RESEARCH vorgestellt werden, dessen Architektur so konzipiert worden ist, daß es Parallelismus auf mehreren Ebenen gleichzeitig ausnutzen kann.

3.3 MULTITASKING AUF DER CRAY X-MP/22

Die Ausnutzung des Parallelismus zur Leistungssteigerung steht schon seit einiger Zeit im Mittelpunkt der Entwicklung neuer Rechnerarchitekturen. Das Vorhandensein mehrerer Funktionseinheiten oder das Pipelining sind nur einige Beispiele für Konzepte, die dazu dienen, die Ausführungszeit von Programmen zu reduzieren. Diesen Konzepten, die schon in den CRAY Rechnern vom Typ CRAY-1 integriert waren, wurde mit der Einführung der Supercomputer vom Typ CRAY X-MP das Konzept des Multitasking hinzugefügt.

3.3.1 Beschreibung der CRAY X-MP/22 Architektur

Die CRAY X-MP/22 ist ein Rechnersystem, das aus **zwei** identischen und symmetrischen **Vektorprozessoren** besteht, die über einen gemeinsamen Speicher und gemeinsame Register **stark** miteinander **gekoppelt** sind. Diese Architektur ermöglicht sowohl simultane Skalar-/Vektorverarbeitung unabhängiger Jobs als auch simultane Skalar-/Vektorverarbeitung unabhängiger Tasks desselben Jobs [Lar,84b]. Weitere Eigenschaften dieses Multiprozessorsystems sind (Abb. 8):

- äußerst **niedrige** Taktzeit von 9.5 ns,
- Ausführung des gemeinsamen Speichers in **bipolarer ECL-Technologie** mit einer Zykluszeit von 38 ns,
- 13 **unabhängige Funktionseinheiten** je CPU, die nach dem Pipeline-Prinzip arbeiten und Register-Register Operationen ausführen,
- **verschiedene Registersätze** für Adressen, Skalare und Vektoren (dabei kann jedes Vektorregister bis zu 64 Skalar-Werte aus dem Speicher enthalten),

- **vier** Speicheranschlüsse (**memory ports**) je CPU, die es ermöglichen, zwei Lese- und eine Schreiboperation von den Vektorregistern zum Speicher sowie unabhängige E/A-Operationen gleichzeitig auszuführen,

- physikalisch und logisch vollständig **getrenntes E/A-Subsystem**,

- Inter-Prozessor **Kommunikations- und Kontroll-Hardware**.

3.3.2 Inter-Prozessor Kommunikationssektion

Die Inter-Prozessor Kommunikationssektion der CRAY X-MP/22 (Abb. 9) enthält spezielle Hardware für eine Echtzeit-Uhr sowie für die Kommunikation und Synchronisation beider CPUs. Die Echtzeit-Uhr wird über das *Echtzeit-Uhrregister* von beiden CPUs gemeinsam benutzt. Beiden Prozessoren stehen gemeinsame Daten- und Synchronisationsregister zur Verfügung, die eine effiziente und koordinierte Verwendung beider Prozessoren für einen Job ermöglichen.

Drei identische Gruppen (**cluster**) gemeinsamer Register sind für die Kommunikation und Synchronisation verfügbar [CRAY,32]. Jede Gruppe besteht aus

8 gemeinsamen 24-Bit **Adressregistern** (SB),
8 gemeinsamen 64-Bit **Skalarregistern** (ST) und
32 gemeinsamen **binären Semaphorregistern** (SM).

Die Semaphorregister ermöglichen beiden CPUs den Austausch von Signalen, die anzeigen, ob die Bearbeitung einer Task beginnen kann, warten muß oder beendet worden ist. Auf den Semaphorregistern ist eine *Test-and-Set*-Instruktion definiert, die eine Sperrfunktion enthält. Diese Sperrfunktion schützt davor, daß beide CPUs die Test-and-Set-Instruktion gleichzeitig ausführen. Das betreffende Semaphorregister wird bei Ausführung der Test-and-Set-Instruktion zuerst getestet. Hat es den Wert 0, erhält es den Wert 1, und die Instruktion endet. Ist der Wert des Registers 1, wird die Instruktion solange angehalten, bis das Register wieder den Wert 0 hat.

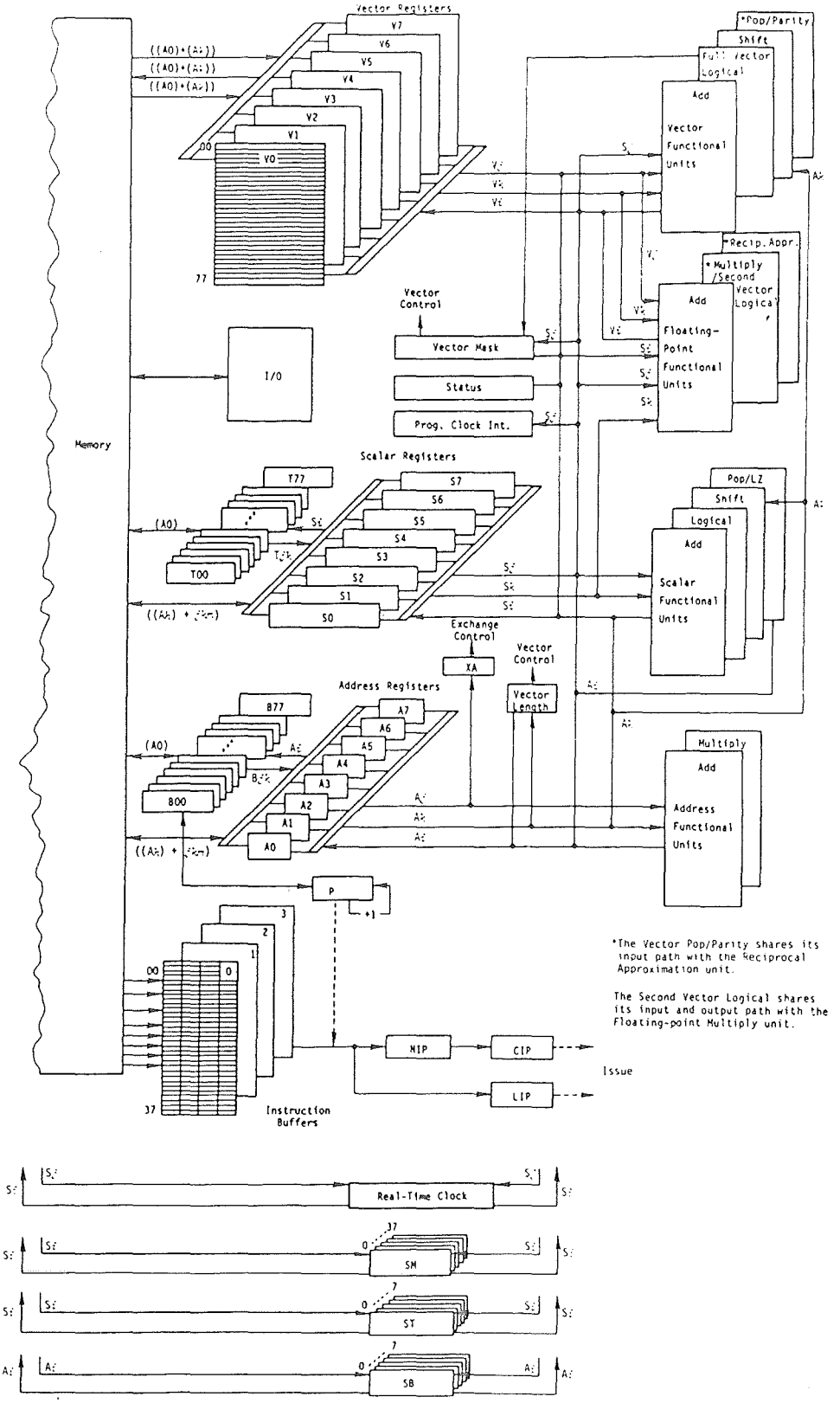


Abb. 8. Kontroll- und Datenwege für eine CPU der CRAY X-MP/22.

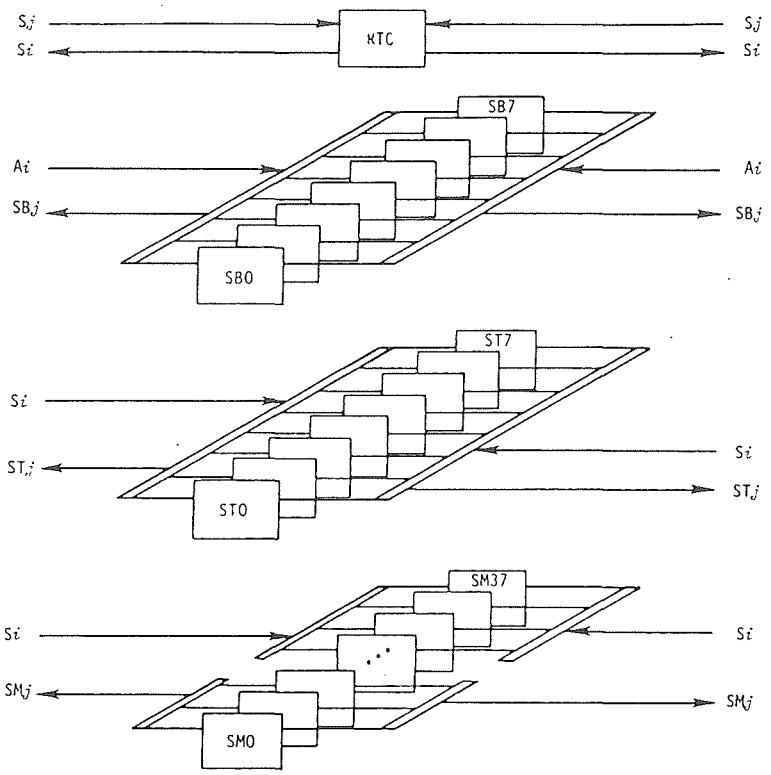


Abb. 9. Echtzeit-Uhr und gemeinsame Register der CRAY X-MP/22.

Die gemeinsamen Adress- und Skalarregister dienen dazu, zwischen den Prozessoren einen schnellen Adressen- und Variablenwerteaustausch zu ermöglichen. Ist ein eingeschränkter Zugriff auf diese Register erforderlich, müssen die Semaphorregister zu Hilfe genommen werden [CRAY,32].
Die Gruppen gemeinsamer Register werden den Prozessoren mit Hilfe des Betriebssystems zugeteilt. Jede CPU besitzt ein *Gruppennummernregister* (cluster number register), dessen Wert anzeigt, welche Gruppe von dieser CPU benutzt wird. Jeder Prozessor kann sowohl im Benutzer- als auch im Betriebssytem-Modus auf eine ihm zugeteilte Gruppe gemeinsamer Register zugreifen.

Nach diesem Hardware-Überblick soll nun untersucht werden, welche Software-Mittel zur Verfügung stehen, um mit Hilfe der Multitasking-Fähigkeiten der CRAY X-MP/22 den in Algorithmen enthaltenen Parallelismus optimal ausnutzen zu können.

3.3.3 Multitasking-unterstützende System-Software

Um Multitasking zu ermöglichen, mußte die schon vorhandene System-Software der Firma CRAY Research modifiziert und neue Software entwickelt werden. Neben der Veränderung des Betriebssystems und des FORTRAN-Compilers wurde eine neue Bibliothek geschaffen, die die Verwaltung und Interaktionen paralleler Tasks unterstützt. Diese Bibliothek wurde in die Anwenderbibliothek UTLIB integriert.
Das CRAY-Betriebssytem (COS) sieht Multitasking zur Zeit nur auf Unterprogramm-Ebene vor. Es gestattet einem Programm unter Verwendung der vorhandenen Bibliotheksroutinen zusätzliche, zu diesem Job gehörende Tasks zu erzeugen. Das Hauptprogramm stellt dabei eine übergeordnete Anfangs-Task (initial task) dar. Unter einer Task versteht man bei CRAY RESEARCH eine Berechnungseinheit (Haupt- oder Unterprogramm), die als Scheduling-Einheit von der Bibliothek verwaltet wird [CRAY,222].
COS ermöglicht sowohl **Einbenutzer-** als auch **Mehrbenutzer-Betrieb**. Im Einbenutzer-Betrieb stehen einem Multitasking-Programm während dessen Ausführung der gesamte Hauptspeicher sowie beide CPUs zur Verfügung. Umfangreiche und vor allem zeitkritische Programme erzielen mit Multitasking in diesem Modus den größten Zeitgewinn. Gleichzeitig verspricht dieser Betriebsmodus die höchste Effizienz. Wird im Einbenutzer-Betrieb die zu leistende Arbeit nicht balanciert auf die parallelen Tasks (und damit auf die Prozessoren) verteilt, dann hat das eine schlechte Auslastung des Systems zur Folge. Im Mehrbenutzer-Betrieb wird ein Multitasking-Programm zusammen mit anderen Programmen ausgeführt. Das können sowohl Programme mit Multitasking als auch Programme ohne Nutzung der Multitasking-Fähigkeiten sein. Eine Verbesserung der Ausführungszeit wird hierbei aufgrund der anderen Programme im System von Lauf zu Lauf stark variieren, weil man nicht vorhersagen kann, wie erfolgreich dieses Multitasking-Programm in der Konkurrenz um die Betriebsmittel des Systems

sein wird (es sei denn, es hat immer höchste Priorität). Der **Durchsatz** des Systems (Anzahl verarbeiteter Instruktionen und Daten pro Zeiteinheit) wird in diesem Betriebsmodus etwas geringer, weil die zusätzlich zu leistende Arbeit (Overhead) des Multitasking-Programms die verfügbare CPU-Zeit für die anderen Programme reduziert. Die Gefahr einer Auslastungsverschlechterung der Prozessoren ist hier nicht gegeben, weil beim Untätigwerden eines Prozessors diesem durch das Betriebssystem ein anderes Programm zur Ausführung zugeteilt werden kann. Die Ausführung von Multitasking-Programmen, die sich in der Entwicklungs- und Experimentierphase befinden, kann für diesen Modus geeignet sein [Lar,84b].

Die Ausführung eines Programms unter Ausnutzung der Multitasking-Fähigkeiten kann sogenannten **reentrant** Objektcode erfordern. Ein Programmodul heißt *reentrant*, wenn es von mehreren Tasks gleichzeitig ausgeführt werden kann. Um dies zu erreichen, müssen der Code und die Daten (lokalen Variablen) eines reentrant Programmoduls getrennt abgelegt werden. Dadurch ist es möglich, jeder Task, die dieses Programmodul ausführt, getrennte Speichersegmente zuzuteilen [Holt,78]. Die Forderung nach reentrant Objektcode läßt sich jedoch umgehen, indem der Code eines von mehreren Tasks auszuführenden Unterprogramms jeweils unter anderem Namen mehrfach in das Multitasking-Programm eingebaut wird.
Der CRAY FORTRAN (CFT) Compiler wurde nun in der Weise verändert, solchen reentrant Objektcode durch Compilation mit der Option ALLOC = STACK erzeugen zu können. Die lokalen Variablen eines reentrant Unterprogramms werden auf einem Stack abgelegt. Jeder Task, die dieses Unterprogramm aufruft, wird ein getrennter Task-Stack zugewiesen, auf dem die lokalen Variablen dann abgelegt werden.
Im Gegensatz dazu steht **nicht-reentrant** oder **seriell wiederverwendbarer** Objektcode. Ein Programmodul, dessen Objektcode *seriell wiederverwendbarer* ist, kann zwar auch von mehreren Tasks gemeinsam benutzt werden, jedoch muß eine Ausführung des Objektcodes beendet sein, bevor die nächste beginnen kann, d.h. die gleichzeitige Ausführung dieses Moduls ist hierbei ausgeschlossen.

Eine Bibliothek von Multitasking-Routinen schafft eine Benutzerschnittstelle zu den Multitasking-Fähigkeiten des COS und der Hardware. Diese Multitasking-Bibliothek bearbeitet die Anforderungen eines Benutzerprogramms und "scheduled" die Tasks dieses Programms oft ohne Interventionen

des Betriebssytems. Das Grundkonzept in der Betriebssystem-Schnittstelle zum Bibliothek-Scheduler ist das der logischen (virtuellen) CPU. Eine logische CPU ist die Scheduling-Einheit des Betriebsystems zur Ausführung auf einer physikalischen CPU [CRAY,222]. Bei Ausführungsbeginn wird jedem Multitasking-Job vom Betriebssystem eine logische CPU zugeteilt. Dem Bibliothek-Scheduler obliegt es dann, während der Ausführung weitere logische CPUs für diesen Job anzufordern. Eine logische CPU wird immer dann benötigt, wenn eine neue Task initiiert wird und keine logische CPU frei ist. Hat das Betriebssytem weitere logische CPUs zugeteilt, besteht die Aufgabe des Bibliothek-Schedulers darin, sie den erzeugten Tasks möglichst effizient zuzuordnen. Muß eine Task beispielsweise auf ein Ereignis warten, wird sie von ihrer logischen CPU getrennt und kommt in eine Warteschlange. Damit ist diese logische CPU dann frei für eine andere Task des Jobs oder für die Rückgabe an das Betriebssystem.
Bei der Implementierung des Bibliothek-Schedulers wurde das Konzept des Monitors verwirklicht. Die wichtigsten parametrisierten Prozeduren dieses Monitors sind die Routinen TSKSTART, TSKWAIT sowie die LOCK- und EVENT-Routinen, deren Ausführung jeweils wechselseitig ausgeschlossen ist. Die Parameter dieser Routinen (Task-Kontroll-Array, Event- bzw. Lock-Variablen) sind auch gleichzeitig globale Variablen des Monitors. Sie werden einmal zu Beginn jeder Ausführung initialisiert. Diese Parameter bzw. globalen Variablen stellen darüberhinaus die Bedingungsvariablen des Monitors dar. Für jede Routine, deren Ausführung eine Blockierung der aufrufenden Task zur Folge haben kann (TSKWAIT, LOCKON, EVWAIT), gibt es jeweils sogenannte Suspendiert-Warteschlangen. Die Tasks werden aufgrund ihrer Zustandsveränderungen, die sich aus der Verwendung der Multitasking-Routinen ergeben, zwischen den Warteschlangen hin- und herbewegt. Im allgemeinen arbeiten alle Warteschlangen nach dem FIFO-Prinzip. Ruft eine Task jedoch irgendeine der Bibliotheksroutinen auf, dann wird sie immer am Kopf der Bereit-Warteschlange (Waiting for Logical CPU queue) eingefügt. Wird eine Task von der Bereit-Warteschlange zu einer Suspendiert-Warteschlange gebracht, so ändert sich bei diesem Übergang ihr Zustand von bereit nach blockiert (Abb. 10).

Nachdem bisher lediglich die Realisierung und die Aufgaben der Multitasking-Bibliothek behandelt wurden, soll nun erläutert werden, wie ein Anwendungsprogrammierer die Routinen dieser Bibliothek nutzen kann.

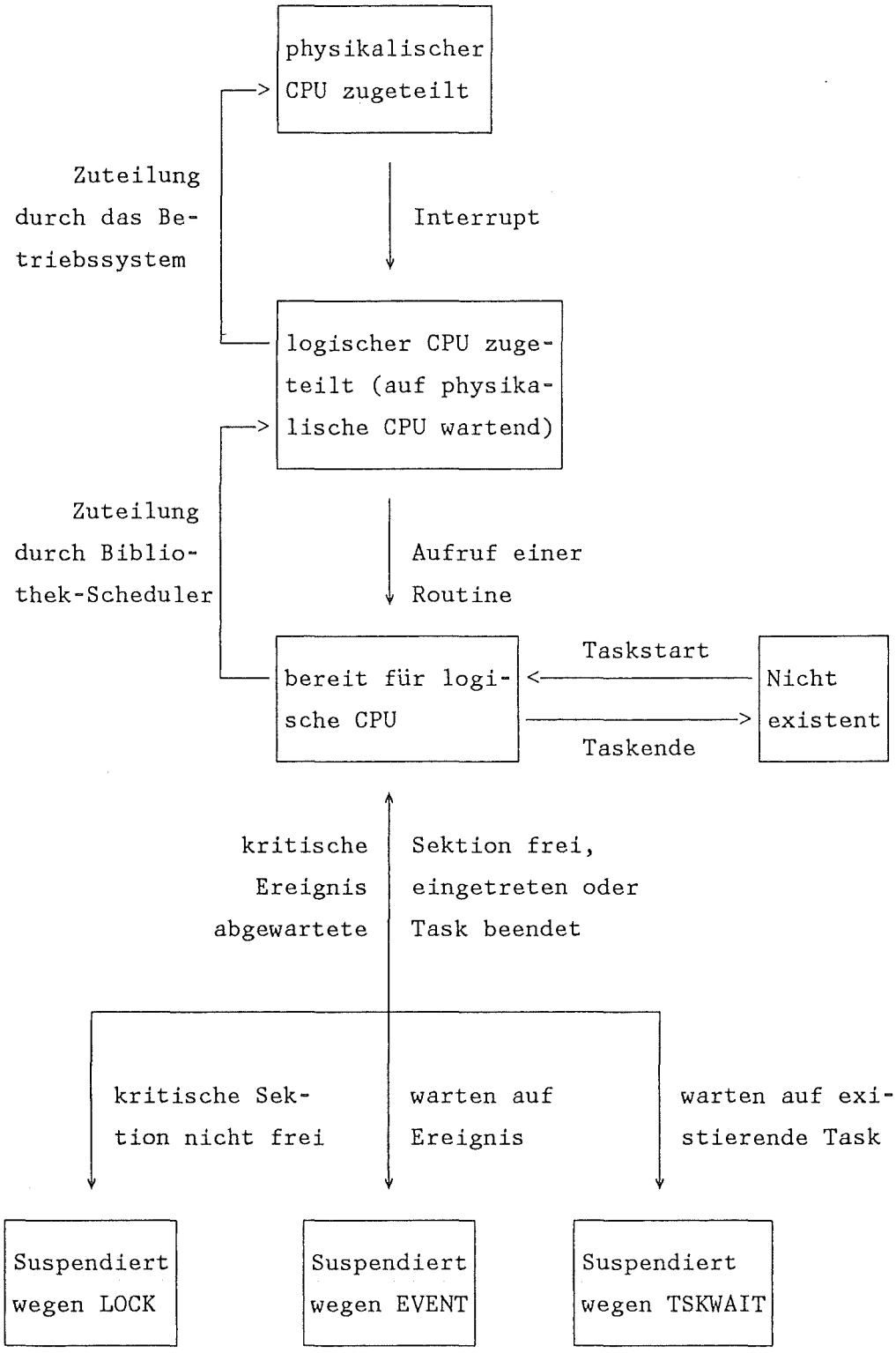


Abb. 10. Zustandsübergänge paralleler Tasks

3.3.4 Multitasking-Programmierung

Die Implementierung des Multitasking bei der Firma CRAY RESEARCH ist auf umfangreiche und zeitkritische FORTRAN-Programme ausgerichtet, die dem ANSI-Standard FORTRAN 77 (American National Standards Institute) entsprechen. Um die Multitasking-Fähigkeiten der System- und Bibliotheksoftware nutzen zu können, braucht ein Benutzer lediglich CALL-Anweisungen der Bibliotheksroutinen in sein FORTRAN-Programm einzubauen. Die Implementierung auf FORTRAN-Ebene ermöglicht neben ihrer Flexibilität und der einfachen Programmierung auch die Transportabilität zu anderen CRAY Rechnersystemen (in eingeschränkter Weise auch zu den Einprozessor-Konfigurationen).
Die Grundprobleme des Multiprocessing werden durch drei Kategorien von Routinen gelöst. Die erste Kategorie ist für die Spezifikation paralleler Ausführung zuständig. Die Aufgabe der zweiten Kategorie besteht darin, die Kommunikation und Synchronisation paralleler Tasks zu bewerkstelligen. Das Problem des wechselseitigen Ausschlusses kritischer Sektionen wird mit Hilfe der dritten Kategorie gelöst.

3.3.4.1 Taskinitiierung und Taskabschluß

Ein Multitasking-Programm erzeugt eine unabhängige parallele Task mit Hilfe der TSKSTART-Routine, deren Aufruf folgendes Aussehen hat:

CALL TSKSTART (*Tskid1, Name*[, *Liste*])

Jeder Task ist ein Task-Kontroll-Array *Tskid1* vom Typ Integer zugeordnet, das aus zwei (optional drei) Komponenten besteht und für die Bibliothek zum Zweck der Task-Identifikation benötigt wird. *Name* gibt den Namen der Subroutine an, die als selbständige Task auszuführen ist. *Liste* bezeichnet eine optionale Liste von Argumenten, die der erzeugten Task als Parameter übergeben werden können. Ein TSKSTART-Aufruf der oben vorgestellten Syntax ist also identisch mit einem CALL *Name* [(*Liste*)] in FORTRAN, mit

der Ausnahme, daß die Subroutine *Name* als selbständige Task ausgeführt wird.

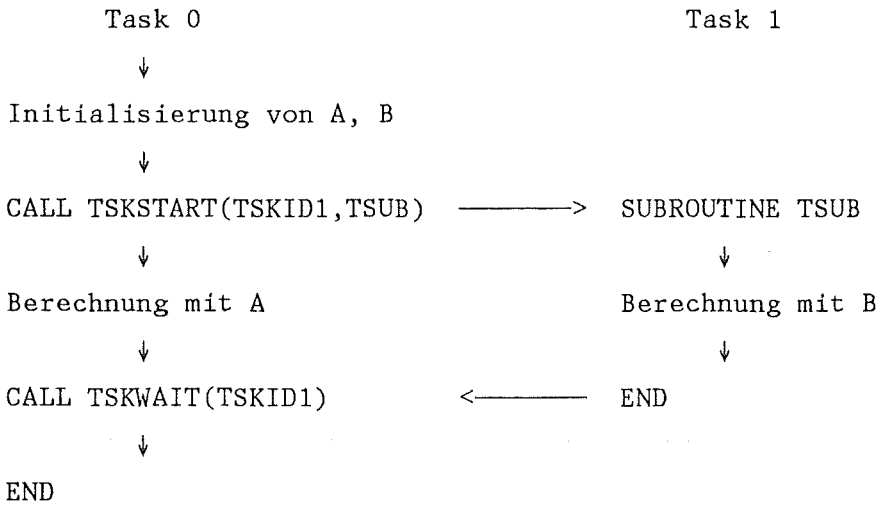
Eine Task wartet auf die Beendigung einer beliebigen Task (muß nicht eine von ihr erzeugte sein) mit Hilfe der TSKWAIT-Routine. Diese Routine wird durch die CFT-Anweisung

```
CALL TSKWAIT (Tskid1)
```

gestartet. Das Task-Kontroll-Array *Tskid1* muß die gleiche Information wie nach dem TSKSTART enthalten. Jene Task, die TSKWAIT aufruft, wird solange blockiert (kommt in die Suspendiert-Warteschlange der TSKWAIT-Routine), bis die zu beendende Task ihre Ausführung abgeschlossen hat. Nach Abschluß dieser Task geht die Kontrolle an die wartende Task zurück.

Beispiel 3.4

Initiierung und Beendigung paralleler Tasks. Wichtig hierbei ist, daß die als Task 1 initiierte Subroutine TSUB in Task 0 als EXTERNAL vereinbart wurde.



Daneben gibt es weitere nützliche Routinen, die der Task-Identifikation (TSKVALUE), der Frage nach der Existenz einer Task (TSKTEST) und der

Frage nach den Zuständen einer Task (TSKLIST) während der Ausführung dienen.

3.3.4.2 Synchronisation mit EVENT-Routinen

Die von einem Programm erzeugten Tasks müssen während ihrer parallelen Ausführung möglicherweise auf das Eintreten eines Ereignisses warten. Für die dadurch bedingte Synchronisation stehen einem Benutzer die sogenannten EVENT-Routinen zur Verfügung. Die Zustände eines Events werden mit Hilfe einer ganzzahligen Variablen (Event-Variablen) festgehalten. Durch den Aufruf

CALL EVASGN (*Event1*[, *Wert*])

wird die ganzzahlige Variable *Event1* für den weiteren Verlauf der Ausführung als Event-Variable identifiziert. Optional kann diese Event-Variable über den Parameter *Wert* auf einen bestimmten Anfangswert gesetzt werden. Alle Tasks, die auf das Berechnungsergebnis einer anderen Task warten müssen, tun dies durch Aufruf der EVWAIT-Routine

CALL EVWAIT (*Event1*),

wobei *Event1* eine Event-Variable bezeichnet. Jede dieser Tasks wird solange blockiert, bis die das Ergebnis liefernde Task mit der EVPOST-Routine durch

CALL EVPOST (*Event1*)

signalisiert, daß das gewünschte Ergebnis vorliegt (*Event1* bezeichnet die gleiche Event-Variable). Die liefernde Task kann nach kurzer Verzögerung in ihrer Ausführung fortfahren. Alle auf das Signal wartenden Tasks werden aus der Suspendiert-Warteschlange herausgeholt und zur Bereit-Warteschlange gebracht. Die Kontrolle wird an den Bibliothek-Scheduler übergeben, der den nun zur Ausführung bereiten Tasks logische CPUs zuteilen kann.

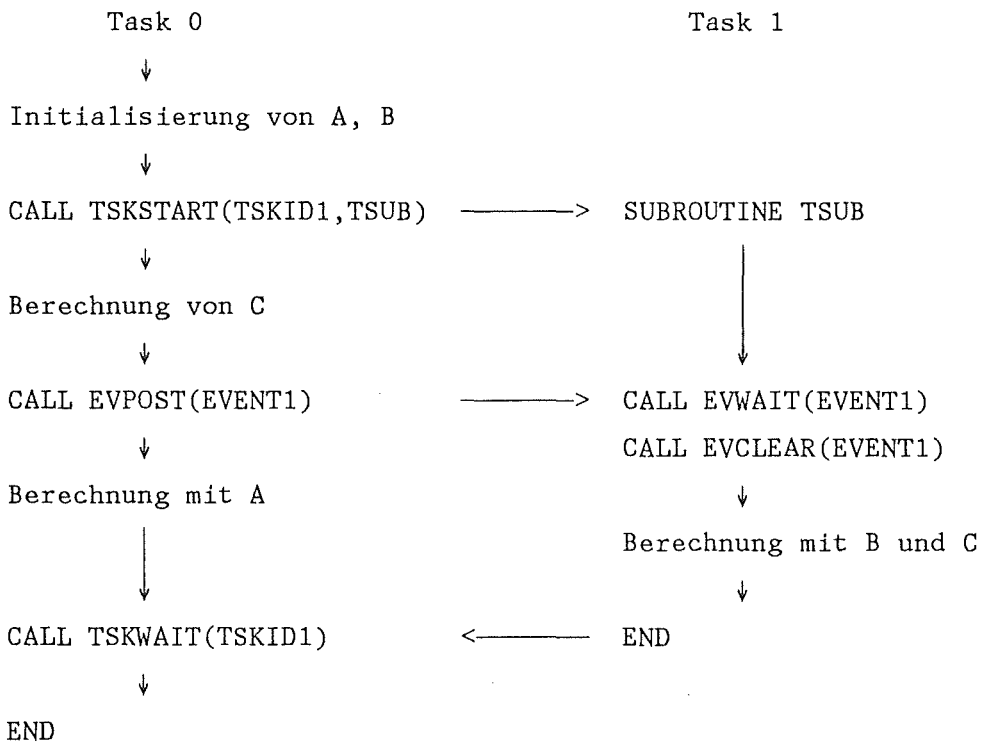
Nachdem das gewünschte Ergebnis nun vorliegt, d.h. die Kommunikation und Synchronisation abgeschlossen worden ist, kann mit der EVCLEAR-Routine durch

CALL EVCLEAR (*Event1*)

die Event-Variable *Event1* wieder gelöscht werden. Die Kontrolle geht an die Task zurück, die EVCLEAR aufgerufen hat. Der Aufruf sollte durch eine der Tasks erfolgen, die auf das Eintreten des Ereignisses gewartet haben. Um eine Event-Variable löschen zu können muß allerdings sichergestellt sein, daß alle Tasks, die das Ergebnis benötigten, die EVWAIT-Routine aufgerufen haben. Wird ein EVCLEAR zu früh aufgerufen, kann das zeitabhängige Fehler und unter Umständen auch einen Deadlock zur Folge haben.

Beispiel 3.5

Synchronisation paralleler Tasks mit EVENT-Routinen.



Wird eine Event-Variable *Event1* nicht mehr benötigt, kann sie mit der EVREL-Routine durch

CALL EVREL (*Event1*)

wieder freigegeben werden. Ist eine Task bei Ausführung von EVREL wegen der Event-Variablen *Event1* blockiert, dann hat das einen Fehler und den Abbruch der Ausführung zur Folge.
Mit Hilfe der EVTEST-Routine läßt sich während der Ausführung prüfen, ob eine Event-Variable durch EVPOST gesetzt worden ist.

3.3.4.3 Behandlung kritischer Sektionen

Nachdem bisher die Initiierung und der Abschluß sowie die Kommunikation und Synchronisation paralleler Ausführung behandelt wurde, interessiert es nun noch, wie der exklusive Zugriff auf gemeinsam benutzte Variablen paralleler Tasks gehandhabt werden kann.

Die Überwachung kritischer Sektionen wird bei CRAY RESEARCH mit Hilfe von **Locks** gelöst, deren Zustände ebenfalls mit Hilfe ganzzahliger Variablen (Lock-Variablen) festgehalten werden.
Durch den folgenden Aufruf der LOCKASGN-Routine

CALL LOCKASGN (*Lock1*[, *Wert*])

wird für den weiteren Verlauf eines Programms *Lock1* als Lock-Variable identifiziert (optional kann sie *Wert* als Anfangswert erhalten).
Das Eintritt-Protokoll einer kritischen Sektion besteht aus dem Aufruf der LOCKON-Routine. Zum Eintritt in ihre kritische Sektion führt eine Task

CALL LOCKON (*Lock1*)

aus. Die Lock-Variable *Lock1* wird auf einen bestimmten Wert gesetzt, der anzeigt, daß die Lock-Variable gesperrt ist. Die Kontrolle geht an die aufrufende Task zurück, d.h. sie kann in ihre kritische Sektion eintreten.

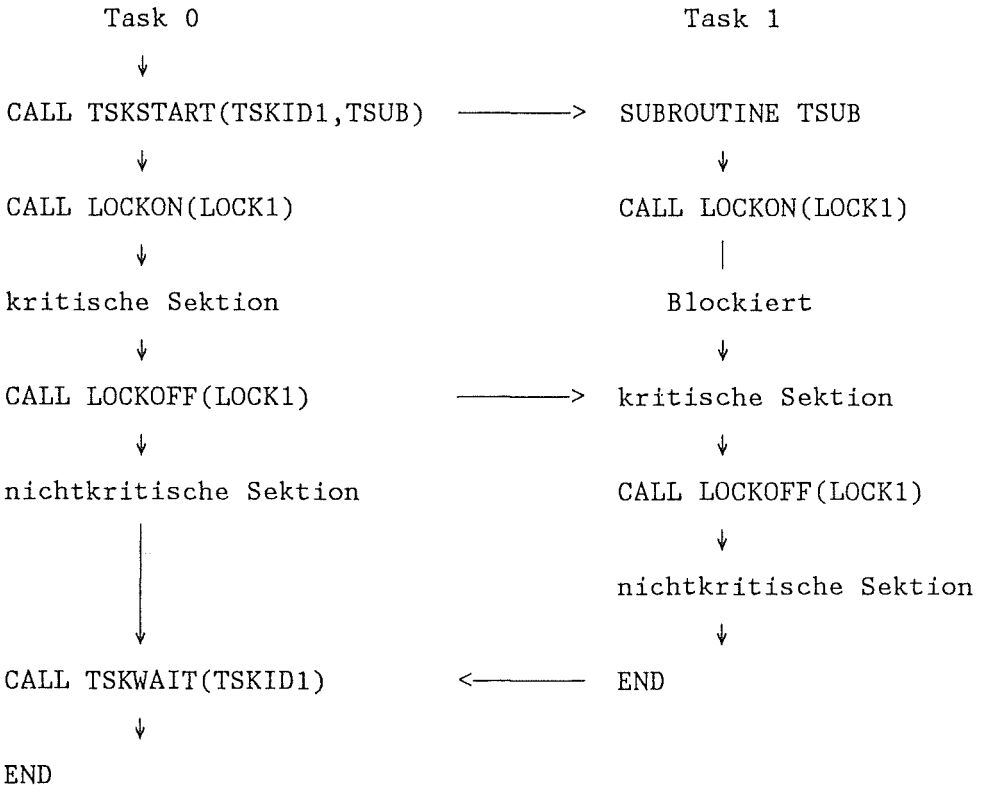
War die Lock-Variable jedoch bereits gesperrt, wird die aufrufende Task solange blockiert, bis die Sperre von einer anderen Task, die sich in ihrer kritischen Sektion befand, durch die LOCKOFF-Routine wieder aufgehoben wird. Es ist klar, daß dadurch wechselseitiger Ausschluß kritischer Sektionen gewährleistet ist.
Das Austritt-Protokoll einer kritischen Sektion sieht wie folgt aus:

```
CALL LOCKOFF (Lock1)
```

Die Kontrolle geht an die aufrufende Task zurück, wodurch sie dann Anweisungen in ihrer nichtkritischen Sektion ausführen kann. Die erste Task, die wegen der Lock-Variablen *Lock1* blockiert war, wird von der Suspendiert-Warteschlange zur Bereit-Warteschlange gebracht, wo sie durch den Bibliothek-Scheduler einer logischen CPU zugeteilt werden kann.

Beispiel 3.6

Wechselseitiger Ausschluß kritischer Sektionen mit LOCK-Routinen.



Wird eine Lock-Variable *Lock1* nicht mehr benötigt, kann sie mit

CALL LOCKREL (*Lock1*)

wieder freigegeben werden. Ist nach Ausführung dieser Routine eine der Tasks wegen der Lock-Variablen *Lock1* blockiert, kommt es zu einem Fehler. Soll während der Ausführung festgestellt werden, ob eine Lock-Variable gesperrt ist, kann das durch Aufruf der LOCKTEST-Routine erreicht werden. Ist die Lock-Variable frei, dann wird sie durch LOCKTEST gesperrt, und die betreffende Task kann beispielsweise in eine direkt folgende kritische Sektion eintreten. Ist die Lock-Variable bei dem Test gesperrt, wird die aufrufende Task nicht blockiert und kann somit in ihrer Arbeit fortfahren.

3.3.4.4 Abstimmung durch den Programmierer

Die Multitasking-System-Software erlaubt einem Benutzer die Ausführung seines Programms selbst abzustimmen. *Abstimmung* (tuning) besagt, daß die Anzahl und Verfügbarkeit logischer CPUs den Anforderungen eines Programms effizient angepaßt werden sollte. Die Abstimmung erfolgt mit Hilfe einer weiteren Monitorprozedur und bestimmter Parameter, die vom Bibliothek-Scheduler verwaltet werden und zu Beginn jeder Ausführung einen Standardwert erhalten. Durch Aufruf der TSKTUNE-Routine hat ein Programmierer die Möglichkeit, die Werte dieser Parameter zu verändern. TSKTUNE wird wie folgt aufgerufen:

CALL TSKTUNE ($Schlüsselwort_1$, $Wert_1$, $Schlüsselwort_2$, $Wert_2$, ...)

Jedes *Schlüsselwort* muß eine Zeichenfolge und jeder dazugehörige *Wert* eine ganze Zahl sein. Die *Schlüsselworte* dürfen folgende Zeichenfolgen umfassen [CRAY,222]:

MAXCPU steht für die erlaubte, maximale Anzahl logischer CPUs eines Jobs (Defaultwert = 2).

DBRELEAS bezeichnet eine stille Reserve logischer CPUs. Sind einem Job mehr logische CPUs zugeordnet als dieser Tasks hat, dann gibt DBRELEAS die maximale Zahl logischer CPUs an, die zurückbehalten werden. Alle anderen logischen CPUs werden an das Betriebssystem zurückgegeben (Defaultwert = 1).

DBACTIVE stellt eine stille Reserve aktiver logischer CPUs dar. Ihr Wert (Default = 0) ist die Anzahl zusätzlicher Tasks, die zur Ausführung bereit gemacht werden können, bevor eine weitere logische CPU aktiviert oder angefordert wird.

HOLDTIME gibt die Zahl der CPU-Zyklen an, die eine logische CPU festgehalten wird, bevor der Bibliothek-Scheduler sie an das Betriebssystem zurückgibt (Defaultwert = 100000 CPU-Zyklen).

SAMPLE steht für die Zahl an CPU-Zyklen zwischen zwei Überprüfungen der Bereit-Warteschlange. SAMPLE reguliert die Häufigkeit der Überprüfungen der Bereit-Warteschlange, wenn eine logische CPU auf die Ausführungsbereitschaft einer Task wartet (Defaultwert = 500 CPU-Zyklen).

Ein Programmierer muß sich jedoch darüber im klaren sein, daß die Werte der Parameter im Einbenutzer-Betrieb von denen des Mehrbenutzer-Betriebs verschieden sein sollten.

- Für eine hohe Effizienz und eine gute Auslastung sollte der Wert von MAXCPU im Einbenutzer-Betrieb entweder gleich der Anzahl physikalischer CPUs oder gleich der Anzahl physikalischer CPUs plus eins gesetzt werden. Durch den letzteren Wert kann beispielsweise dann eine Verbesserung erzielt werden, wenn in einem Programm eine Task E/A-Anweisungen ausführt und zwei andere Tasks Berechnungen leisten. Im Mehrbenutzer-Betrieb dagegen sollte der Wert von MAXCPU klein gehalten werden, weil dort die logischen CPUs von verschiedenen Jobs benötigt werden.

- Mit dem Ziel einer hohen Effizienz können die letzten vier Parameter auf einer dedizierten Maschine höhere Werte haben, weil die Kosten nicht genutzter CPU-Zyklen geringer sind als die Kosten der ständigen

Kontaktierung des Betriebssytems. Beispielsweise sollte DBRELEAS auf MAXCPU - 1 gesetzt werden, damit unbenutzte logische CPUs nicht zu schnell an das Betriebssystem zurückgegeben werden.

- Im Mehrbenutzer-Betrieb sind kleine Werte (die Defaultwerte) der letzten vier Parameter erwünscht, weil die verschwendeten CPU-Zyklen die Auslastung der Prozessoren und den Durchsatz des Systems verringern würden.

3.4 PROBLEME MIT MULTITASKING

Während der Ausführung eines Multitasking-Programms kann nicht vorherbestimmt werden, in welcher Reihenfolge die Tasks dieses Programms ausgeführt werden. Die Ausführungsreihenfolge und die Beendigung der Tasks eines Jobs sowie die Verfügbarkeit der Prozessoren sind Bestandteile der Scheduling-Politik des Bibliothek-Schedulers und des Betriebssystems. Die Aufgabe eines Programmierers besteht nun darin, diesen zeitlichen Nichtdeterminismus in einen ergebnisbezogenen Determinismus zu überführen. Erschwert wird diese Anforderung dadurch, daß die Fehlersuche in Multitasking-Programmen nicht so einfach wie in sequentiellen Programmen ist. Wegen des zeitlichen Nichtdeterminismus können zeitabhängige Fehler auftreten, die nicht reproduzierbar sein können und wegen (noch) nicht ausreichend vorhandener Hilfsmittel nur äußerst mühsam zu entdecken und zu verhindern sind. Eine fehlerfreie Programmausführung wird dann erreicht, wenn aus der genauen Kenntnis der parallelen Abläufe eines Programms die Multitasking-Bibliotheksroutinen präzise in das Programm eingebaut werden können, um damit die Kommunikation und Synchronisation der parallelen Tasks sowie den Schutz gemeinsam benutzter Variablen sicherzustellen.

3.4.1 Bereiche von Variablen

Um die korrekte parallele Ausführung eines Programms zu gewährleisten, muß der **Bereich** jeder Variablen dieses Programms analysiert werden.
Unter dem *Bereich* einer Variablen versteht man jenen Teil eines Programms, in dem diese Variable *definiert* ist und *benutzt* werden kann. Außerhalb ihres Bereichs ist eine Variable nicht definiert, und Zugriffe auf diese Variable beziehen sich entweder wie in FORTRAN auf eine andere Variable gleichen Namens oder führen wie in PASCAL zu einem Fehler.

Jede Task eines Multitasking-Programms besteht aus ausführbaren Anweisungen und einer wohldefinierten Menge von Variablen, auf denen diese Anweisungen arbeiten. Diese Menge kann in zwei Teilmengen unterteilt werden: eine Menge **lokaler** und eine Menge **globaler** (COMMON-) Variablen. Der Bereich lokaler Variablen erstreckt sich nur auf eine Task. Für den Fall, daß die untergeordneten Programmeinheiten (Subroutinen, Funktionen) einer Task lokale Variablen dieser Task benutzen müssen, gibt es im CFT eine FORTRAN Spracherweiterung, die sogenannte TASK COMMON-Anweisung. Alle lokalen Variablen einer Task, die in einem TASK COMMON-Block vereinbart wurden, sind dann für sämtliche von dieser Task aufgerufenen Unterprogramme global. Wenn das auszuführende Programm mit der Option ALLOC = STATIC compiliert wurde, wird TASK COMMON wie gewöhnlicher COMMON behandelt und zur Ladezeit den in einem TASK COMMON-Block vereinbarten Variablen statisch Speicherplatz zugeteilt. Wenn mit ALLOC = STACK übersetzt wurde, wird für TASK COMMON-Blöcke zur Laufzeit bei Initiierung einer Task auf dem Task-Stack dynamisch Speicherplatz angelegt. Bei Beendigung einer Task wird der für einen TASK COMMON-Block benötigte Speicherplatz wieder freigegeben.
Wird eine globale Variable in einer anderen Task benutzt, dann wird sie zur gemeinsamen Variablen. Gemeinsame Variablen sind auch die Lock- und Event-Variablen, d.h. alle Lock- und Event-Variablen müssen in COMMON-Blöcken vereinbart werden. Damit erstreckt sich ihr Bereich auf alle Tasks, in denen sie vereinbart werden.
Mit dem Bereich einer Variablen eng verbunden ist das Konzept der Lebensdauer ihres Inhalts. Während die Werte von COMMON-Variablen für die gesamte Programmausführung erhalten bleiben, sind die Werte lokaler Variablen durch den CFT-Compiler nur für die Lebensdauer der entsprechenden Programmeinheit gesichert. Für ein Programm, das ohne Ausnutzung der

Multitasking-Fähigkeiten ausgeführt wird, ist diese Unterscheidung unerheblich, weil den lokalen Variablen für die Dauer der Ausführung ein fester Platz im Hauptspeicher zugewiesen wird. Im Gegensatz dazu kann sich der Speicherplatz der lokalen Variablen eines Multitasking-Programms während der Ausführung ändern. Besonders wichtig ist dieser Sachverhalt für die Parameterübergabe beim TSKSTART-Aufruf. Wie schon erwähnt, werden lokale Variablen auf einem Stack festgehalten. Der Speicherplatz für den Stack wird jedoch nach Ausführung eines RETURN wiederverwendet, so daß der Speicherplatz der Übergabeparameter bereits wiederverwendet worden sein kann, bevor die erzeugte Task die Werte der Parameter benutzt hat. Deshalb sollten lokale Variablen nicht als Parameter übergeben werden, wenn die erzeugende Task nicht auch das TSKWAIT für die erzeugte Task ausführt.

Neben der sorgfältigen Handhabung lokaler Variablen muß in einem Multitasking-Programm vor allem auf die Bearbeitung gemeinsamer Daten sowie auf die Verwendung der Interaktionsmechanismen geachtet werden, da es sonst zu falschen Ergebnissen und im schlimmsten Fall zu einem Deadlock kommen kann.

3.4.2 Gefahr eines Deadlocks

Der Schutz gemeinsamer Variablen und die Interaktionen paralleler Tasks erfordern den Einsatz derjenigen Bibliotheksroutinen, die die Zustände der Lock- und Event-Variablen verändern. Die fehlerhafte Verwendung dieser Routinen kann zu einem Deadlock führen. Eine Deadlock-Situation tritt beispielsweise durch Ausführung der folgenden Programmsequenz ein:

```
      DO 10 I = 1,N
        CALL LOCKON(LOCK)
        ...
   10 CONTINUE
        CALL LOCKOFF(LOCK)
```

In der ersten Iteration wird die Lock-Variable von der aktiven Task zwar erfolgreich gesetzt, in der zweiten jedoch wird die Task wegen der noch

gesetzten Lock-Variablen LOCKI blockiert. Weil LOCKI nicht durch ein LOCKOFF gelöscht werden kann, kommt diese Task in den Deadlock-Zustand. Befindet sich eine Task im Deadlock-Zustand, wird das ausgeführte Programm abgebrochen. Desgleichen kann ein Deadlock bei geschachtelter Verwendung der LOCK-Routinen auftreten.
Die Benutzung der EVENT-Routinen beinhaltet ebenfalls die Gefahr eines Deadlocks. Ein Deadlock tritt hier beispielsweise dann auf, wenn eine Task durch Aufruf von EVWAIT blockiert wird, aber keine andere Task im Laufe der Programmausführung durch EVPOST signalisiert, daß die blockierte Task weitermachen kann. Spätestens dann, wenn eine andere Task auf die Beendigung der blockierten Task durch Aufruf von TSKWAIT wartet, d.h. auch blockiert wird, befinden sich zwei Tasks im Deadlock-Zustand, der sich auf alle anderen Tasks des Programms ausbreiten kann. Das Programm wird nach Entdeckung des Deadlocks abgebrochen.
Die Deadlock-Entdeckung wird von der System-Software vorgenommen. Ein Benutzer erhält nach Entdeckung eines Deadlocks die Meldung, daß alle Tasks wegen Lock- bzw. Event-Variablen oder auf die Beendigung von Tasks warten müssen, d.h. alle Tasks sind blockiert. Für die Nutzung der Multitasking-Fähigkeiten besteht die Aufgabe eines Programmierers nun darin, diesen Programmierfehler zu beheben, um somit die Entstehung weiterer Deadlock-Situationen zu verhindern.

3.4.3 Overhead der Multitasking-Routinen

Der durch die Verwendung der Multitasking-Bibliotheksroutinen entstehende Overhead ist wie schon erwähnt ein wichtiger Einflußfaktor für die erreichbare Beschleunigung eines Algorithmus. Deshalb soll im folgenden erläutert werden, wie hoch dieser Overhead jeweils ist, und welche Unterschiede bei der Ausführung dieser Routinen auftreten können. Es ist klar, daß diese Betrachtungen i. allg. nur für solche Fälle von Interesse sind, bei denen der Overhead an der Ausführungszeit eines Multitasking-Programms signifikanten Anteil hat.

Der höchste Overhead und die größten Unterschiede ergeben sich bei der Initiierung einer Task durch Aufruf der TSKSTART-Routine. Dabei lassen sich im wesentlichen die folgenden Fälle unterscheiden.

1. TSKSTART *ohne Parameter*: Ist die Dimension der lokalen Variablen in der initiierten Task klein, beispielsweise eine 50 × 50 Matrix, dann beträgt der Aufwand für das erste TSKSTART ca. 62000 Takte, für das zweite ca. 7500 Takte und für jedes weitere ca. 3700 Takte. Dabei ist es unerheblich, ob mit ALLOC = STACK oder mit ALLOC = STATIC compiliert wurde.
   Ist der benötigte Speicherplatz der lokalen Variablen hingegen groß, beispielsweise drei 600 × 600 Matrizen, dann kann das erste TSKSTART bis zu 500000 Takte dauern. Der Aufwand für jedes zusätzliche TSKSTART ist dann von der Zeit abhängig, die benötigt wird, um weiteren Speicherplatz anzufordern. Dieser hohe Overhead tritt jedoch nur dann auf, wenn mit ALLOC = STACK übersetzt wurde. Kann mit ALLOC = STATIC übersetzt werden, d.h. kann auf reentrant Code verzichtet werden, dann läßt sich der Overhead auf ähnlich niedrige Werte wie bei kleiner Dimension der lokalen Variablen reduzieren.

2. TSKSTART *mit Parametern*: Durch die Verwendung von Parametern wird der Overhead etwas erhöht. Bemerkbar macht sich dies sowohl bei Compilation mit ALLOC = STATIC als auch bei ALLOC = STACK. Für jede der Compiler-Optionen betrug der Overhead bei kleiner Dimension der lokalen Variablen beispielsweise bei Übergabe von drei 200 × 200 Matrizen für das erste TSKSTART ca. 67000 Takte. Die Werte der restlichen TSKSTARTs lagen bei ca. 8000 Takten für das zweite und bei ca. 4200 für jedes weitere.
   Für die Option ALLOC = STATIC ist die Höhe des Overheads unabhängig von der Dimension der lokalen Variablen. Wird hingegen mit ALLOC = STACK compiliert, so wird der Overhead wie oben von der Zeit dominiert, die nötig ist, um den für die lokalen Variablen benötigten Speicherplatz anzufordern.

Diese Schwierigkeiten können vermieden werden, wenn man folgendes beachtet. Den Stackspeicherplatz erhält jede Task über einen sogenannten **Heap**. Dieser *Heap* stellt einen Teilbereich des Speicherplatzes dar, der für jeden Job angelegt wird. Die Heap-Verwaltungsroutinen sind nun dafür ver-

antwortlich, daß bei Ausführung eines TSKSTART der benötigte Speicherplatz in Form eines Stacks dynamisch zugeteilt wird. Zur Ladezeit wird der Heap eines jeden Jobs mit einer bestimmten Anfangsgröße initialisiert. Ist zur Laufzeit nicht mehr genügend Speicherplatz vorhanden, müssen die Heap-Verwaltungsroutinen vom Betriebssystem neuen Speicherplatz anfordern. Diese Anforderungszeit schlägt sich in dem hohen Overhead nieder, wenn der benötigte Speicherplatz der lokalen Variablen groß ist. Die JCL (Job-Control-Language) bietet die Möglichkeit, über den MM-Parameter die Größe des Heaps zur Ladezeit so vorzubesetzen, daß diese Anforderung über das Betriebssystem entfallen kann. Die Nutzung dieses Parameters wird i. allg. davon abhängen, ob man im Mehrbenutzer-Betrieb oder auf einer dedizierten Maschine rechnet.
Der Aufwand für die TSKWAIT-Routine beträgt 1500 Takte, wenn die abzuwartende Task noch existiert. Dazu kommt noch die Zeit, die gewartet werden muß, bis die betreffende Task ihre Ausführung beendet hat. Ist die Ausführung bereits abgeschlossen, dann beträgt der Overhead nur ca. 200 Takte. Der Overhead für diese Routine kann jedoch auch sehr hoch sein, wenn bei Beendigung der abgewarteten Task ein hoher Speicherplatz an das Betriebssystem zurückgegeben wird.
Der Overhead für die Routinen TSKVALUE und TSKTEST beträgt ca. 200 Takte.

Während der Overhead für die Verwendung der EVASGN-Routine durchschnittlich bei 200 Takten liegt, treten bei Verwendung von EVPOST und EVWAIT Unterschiede auf. Wenn bei Ausführung von EVPOST keine Tasks warten, beträgt der Overhead 200 Takte. Im anderen Fall liegt er bei etwa 1500 Takten. Deshalb wird es beispielsweise meist günstiger sein, jene Task, die EVPOST aufruft, eine Iteration weniger ausführen zu lassen als alle anderen Tasks, die EVWAIT aufrufen, um somit den entstehenden Overhead zu minimieren. Untersuchungen haben gezeigt, daß der Aufwand für EVPOST auch höher sein kann. Ist die wartende Task mehr als 5000 Takte lang blockiert, kann der Overhead für EVPOST bis zu 2000 Takte betragen.
Ist bei Ausführung von EVWAIT bereits durch EVPOST das Eintreten eines Ereignisses signalisiert worden, hat dies einen Overhead von 200 Takten zur Folge. Muß das Eintreten des Ereignisses noch abgewartet werden, beträgt der Aufwand etwa 1500 Takte plus der Zeit, die auf das EVPOST gewartet werden muß. Der Aufwand für diese Routine kann jedoch auch bis zu 2000 Takte kosten, wenn die wartende Task länger als 5000 Takte blockiert ist.

Für das Löschen einer Event-Variablen mit Hilfe der EVCLEAR-Routine werden 200 Takte benötigt, wenn das EVPOST vor dem EVWAIT ausgeführt wird und ca. 300 Takte im anderen Fall. Der Aufwand für EVREL und EVTEST liegt bei etwa 200 Takten.

Ähnlich wie bei der Verwendung von EVASGN beträgt der Overhead für LOCKASGN auch 200 Takte.
Der Overhead für die Ausführung von LOCKON beträgt 200 Takte, wenn die Lock-Variable frei ist. Sonst liegt er bei etwa 1500 Takten plus der Zeit, die eine Task warten muß, bis die Lock-Variable frei wird.
Warten keine Tasks auf den Eintritt in eine kritische Sektion, werden für die Ausführung von LOCKOFF 200 Takte benötigt. Für den Fall, daß Tasks warten, beträgt der Overhead 1500 Takte. Die Minimierung des Overheads durch geschickte Manipulation der Berechnungen ist hier schwieriger als bei den EVENT-Routinen und wird nur in ganz besonderen Anwendungen möglich sein.
Ähnlich wie bei den EVENT-Routinen beträgt der Overhead für die Ausführung von LOCKREL und LOCKTEST jeweils 200 Takte.

Nachdem bisher gezeigt wurde, mit welchen Mitteln man die Multitasking-Fähigkeiten des Multiprozessorsystems CRAY X-MP/22 nutzen kann und worauf man dabei zu achten hat, soll nun anhand einiger Algorithmen untersucht werden, in welchem Maße die Multitasking-Fähigkeiten zur Beschleunigung dieser Algorithmen beitragen können.

4.0 NUTZUNG DER MULTITASKING-FÄHIGKEITEN FÜR STANDARDALGORITHMEN

Parallele Rechnerarchitekturen bieten die Möglichkeit, Berechnungen parallel auszuführen. Diese Möglichkeit und die Forderung nach Ausführungsbeschleunigung sequentieller Algorithmen führt zur Entwicklung originärer **paralleler Algorithmen**. Anhand dieser parallelen Algorithmen kann die Fähigkeit zur Parallelverarbeitung dieser Systeme effizient ausgenutzt werden. Multitasking auf Unterprogramm-Ebene und Vektorverarbeitung auf Anweisungs-Ebene geben Aufschluß über das hohe Maß an Parallelität, welches das Rechnersystem CRAY X-MP/22 zur Ausführung paralleler Algorithmen anbietet.

Während sich beispielsweise sequentielle Algorithmen häufig in natürlicher Art und Weise auf Algorithmen für SIMD-Rechner abbilden lassen, ist die Entwicklung paralleler Algorithmen für MIMD-Rechner ungleich schwieriger. Um einen sequentiellen Algorithmus parallel auf einem Multiprozessorsystem ausführen zu können, muß dieser meist völlig neu strukturiert werden. Zum anderen muß sichergestellt sein, daß der parallele Algorithmus das gleiche Ergebnis wie der optimale sequentielle Algorithmus liefert. Die Unterschiedlichkeit der Architekturen und die Schwierigkeit der Abbildung eines Algorithmus auf diese Architekturen machen deutlich, warum der Umfang an Entwicklungen und Spezifikationen paralleler Algorithmen für Multiprozessorsysteme noch recht gering ist.

Im folgenden soll beispielhaft anhand von Standardalgorithmen die Leistungsfähigkeit und die Anwendungsmöglichkeiten der bisher vorgestellten Konzepte verdeutlicht werden. In den parallelen Algorithmen wurden zur Spezifikation der Synchronisationsmechanismen die CRAY-Notationen verwendet. Alle aufgeführten Zeitmessungen sind auf einer CRAY X-MP/22 im Einbenutzer-Betrieb, d.h. in dedizierter Form durchgeführt worden.

4.1 KLASSIFIKATION PARALLELER ALGORITHMEN

Ein *paralleler Algorithmus* für ein Multiprozessorsystem ist eine Vorschrift zur Ausführung von K parallelen Tasks, die zur Lösung eines gegebenen Problems gleichzeitig und kooperierend arbeiten. Die Stellen, an denen die Tasks eines parallelen Algorithmus kommunizieren und synchronisieren, nennt man **Interaktionspunkte**. Diese Interaktionspunkte unterteilen einen Algorithmus in **Abschnitte**. Am Ende eines jeden Abschnitts kann eine Task dann mit einer anderen kommunizieren, bevor die Ausführung des nächsten Abschnitts beginnt [Kung,76]. Die Interaktionen paralleler Tasks führen dazu, daß einige Tasks während der Ausführung des öfteren blockiert werden. Muß eine Task T_0 auf das Ergebnis einer Task T_1 warten, bleibt T_0 nach Ausführung von EVWAIT solange im Blockiert-Zustand, bis T_1 durch EVPOST die Beendigung des Berechnungsabschnitts signalisiert.
Parallele Algorithmen, in denen Tasks explizit synchronisieren müssen, werden als **synchrone** parallele Algorithmen bezeichnet. Explizite Synchronisation heißt, daß Tasks auf die Ergebnisse der Berechnungen anderer Tasks **warten müssen**. Weil die Ausführungszeit eines Abschnitts von den Eingabedaten und den Aktivitäten im System abhängt, bedeutet dies, daß alle Tasks, die an einem Interaktionspunkt synchronisieren wollen, auf die langsamste unter ihnen warten müssen. Für die Beschleunigung eines synchronen Algorithmus kann dies jedoch ein erheblicher Nachteil sein, weil die blockierten Tasks nicht zur parallelen Ausführung beitragen. Gleichzeitig kann diese Art von Algorithmen eine schlechte Prozessorauslastung zur Folge haben. Deshalb ist es bei synchronen parallelen Algorithmen besonders wichtig, auf eine ausreichende Granularität der Abschnitte und eine balancierte Verteilung der Arbeit zu achten.

Asynchrone parallele Algorithmen sind Algorithmen, bei denen die parallelen Tasks nicht nach jedem Berechnungsabschnitt synchronisieren müssen. Im Gegensatz zu einem synchronen Algorithmus **müssen** in einem asynchronen Algorithmus die Tasks **nicht** auf Ergebnisse anderer Tasks **warten**, um in der Ausführung fortzufahren. Zwischen den Abschnitten verschiedener Tasks bestehen keine expliziten Präzedenzrelationen. Die Kommunikation und Synchronisation der Tasks untereinander wird in einem asynchronen Algorithmus durch die Verwendung gemeinsamer Variablen bewerkstelligt.

Der exklusive Zugriff auf diese gemeinsamen Variablen ist durch die Verwendung kritischer Sektionen gewährleistet. Dabei beinhaltet beispielsweise die Verwendung von LOCKON und LOCKOFF eine implizite Synchronisation der Tasks. Es ist klar, daß zum Erreichen einer guten Beschleunigung auch bei asynchronen parallelen Algorithmen die Arbeit auf alle Tasks balanciert verteilt werden muß.

Neben den synchronen und asynchronen parallelen Algorithmen gibt es jedoch auch parallele Algorithmen, bei denen die parallelen Tasks sowohl explizit synchronisieren müssen als auch asynchron arbeiten können [Kung 76]. Diese Art von Algorithmen bezeichnet man als **semi-synchrone** oder als **semi-asynchrone** parallele Algorithmen.

4.1.1 Asynchrone parallele Algorithmen

In vielen asynchronen parallelen Algorithmen gibt es gemeinsame Variablen, auf die alle Tasks zugreifen können. Wenn die Ausführung eines Task-Abschnitts beendet worden ist, liest diese Task die Werte der von ihr benötigten gemeinsamen Variablen. Aufgrund dieser Werte und der Ergebnisse des zuletzt ausgeführten Abschnitts ändert die Task zuerst die Werte dieser gemeinsamen Variablen, um anschließend entweder einen nachfolgenden Abschnitt zu aktivieren oder die Ausführung zu beenden. Damit Konflikte bzw. Laufzeitfehler vermieden werden, müssen die Operationen auf den gemeinsamen Variablen in kritische Sektionen eingeschlossen werden. Obwohl das Hauptmerkmal eines asynchronen parallelen Algorithmus darin besteht, daß seine Tasks niemals auf die Ergebnisse anderer Tasks warten müssen, kann es trotzdem vorkommen, daß Tasks während ihrer Ausführung blockiert werden. Dies ist genau dann der Fall, wenn der Eintritt in eine kritische Sektion nicht frei ist. Deshalb ist darauf zu achten, daß die Granularität einer kritischen Sektion so klein wie möglich ist, da sich sonst einige Tasks zu lange im Blockiert-Zustand befinden würden und somit CPU-Zyklen ungenutzt blieben.

4.1.1.1 Algorithmus zur Minimum-Maximum-Suche

Ein einfacher asynchroner paralleler Algorithmus dient der Minimum-Maximum-Bestimmung in einem Array reeller Zahlen. Der Algorithmus kann wie folgt beschrieben werden:
Gegeben sei ein Array A, welches N reelle Zahlen A[1], A[2], ..., A[N] enthält. Gesucht ist das kleinste und größte dieser N Elemente.
Der sequentielle Algorithmus hat folgendes Aussehen:

Algorithmus *Minimum_Maximum*

```
max := A[1]
min := max
for i := 2 to N do
   if A[i] < min then
      min := A[i]
   if A[i] > max then
      max := A[i]
```

Zur Parallelisierung dieses sequentiellen Algorithmus kann die zu leistende Arbeit statisch balanciert auf zwei Tasks so verteilt werden, daß beide Tasks annähernd die gleiche Arbeit zu leisten haben. Der Algorithmus wird dabei so umstrukturiert, daß beide Tasks jeweils eine Hälfte des Arrays A durchsuchen und dabei ein lokales Minimum bzw. Maximum bestimmen. Wurden die lokalen Minima und Maxima bestimmt, muß in einer kritischen Sektion festgestellt werden, welches der beiden das kleinere bzw. das größere ist. Dazu werden zwei gemeinsame Variablen *gmin* und *gmax* benötigt, in denen das globale Minimum bzw. Maximum abgelegt wird. Im weiteren werden für Eintritt- bzw. Austrittprotokoll einer kritischen Sektion die CRAY-spezifischen Routinen LOCKON bzw. LOCKOFF benutzt. Zur Erzeugung einer Task wird die Routine TSKSTART und zu deren Abschluß TSKWAIT verwendet. Task0 bezeichnet im folgenden immer die Haupt-Task eines parallelen Algorithmus. Alle im weiteren vorgestellten Algorithmen (sequentiell und parallel) wurden mit der Option ALLOC = STATIC durch den CFT 1.14-Compiler (BF4) übersetzt und unter COS 1.14 (BF4) ausgeführt.
Der parallele Algorithmus zur Minimum-Maximum-Bestimmung hat dann folgendes Aussehen:

Algorithmus *Minimum_Maximum_Parallel*

```
Task0 führt aus:
   gmax := A[1]
   gmin := gmax
   CALL TSKSTART(TSKID1,Task1)
   iu := N div 2
   lokmax := A[1]
   lokmin := lokmax
   for j := 2 to iu do
      if A[j] < lokmin then
         lokmin := A[j]
      if A[j] > lokmax then
         lokmax := A[j]
   CALL LOCKON(LOCKM)
   if lokmin < gmin then
      gmin := lokmin
   if lokmax > gmax then
      gmax := lokmax
   CALL LOCKOFF(LOCKM)
   CALL TSKWAIT(TSKID1)

Task1 führt aus:
   il := N div 2 + 1
   lokmax := A[il]
   lokmin := lokmax
   for j := il+1 to N do
      if A[j] < lokmin then
         lokmin := A[j]
      if A[j] > lokmax then
         lokmax := A[j]
   CALL LOCKON(LOCKM)
   if lokmin < gmin then
      gmin := lokmin
   if lokmax > gmax then
      gmax := lokmax
   CALL LOCKOFF(LOCKM)
```

Für die Implementierung des sequentiellen und des parallelen Algorithmus wurde das zu durchsuchende Array A mit Zufallszahlen zwischen 0 und 1 initialisiert. Der während der Ausführung dieses parallelen Algorithmus entstehende Overhead setzt sich aus der Zeit zusammen, die für die Erzeugung und den Abschluß der Task1 notwendig ist. Dazu kommt noch ein geringer Anteil, der sich aus der Ausführung der Eintritt- bzw. Austrittprotokolle ergibt. Aus der Struktur des parallelen Algorithmus wird offensichtlich, daß dieser Overhead für unterschiedlich große Datenmengen konstant ist. Bei den Zeitmessungen haben sich lediglich geringe Unterschiede von bis zu 2000 Takten bei der Ausführung des TSKSTART ergeben. Weitere Untersuchungen haben ergeben, daß Task0 ihre Berechnungen immer vor Task1 abgeschlossen hatte und somit das TSKWAIT 1500 Takte plus der Zeit kostete, die auf die Beendigung von Task1 gewartet werden mußte. Ursache dafür war, daß Task1 mit ihrer Ausführung immer nach Task0 begann. Aus diesem Grund wurden die Iterationen so auf beide Tasks verteilt, daß Task1 etwas weniger Arbeit leisten mußte und dadurch vor Task0 fertig war. Dies hatte auch zur Folge, daß in den meisten Fällen nicht beide Tasks die kritischen Sektionen gleichzeitig beanspruchten. Somit konnte der Aufwand für LOCKON bzw. LOCKOFF in beiden Tasks auf durchschnittlich 200 Takte reduziert werden. Mit diesen Veränderungen ergab sich insgesamt ein mittlerer Overhead von 65000 Takten. Darüberhinaus müssen für den parallelen Algorithmus noch Verzögerungen durch auftretende Bankkonflikte berücksichtigt werden. Wenn beide Tasks überlappt auf zwei verschiedene Elemente des Feldes A zugreifen wollen, die auf derselben Speicherbank abgelegt sind, kann dies eine Verzögerung von bis zu 3 Takten bedeuten.

Bei den Untersuchungen der Algorithmen wurde festgestellt, daß der Multitasking-Teil f für beliebige N nahezu 100% beträgt (gemessen wurden 99.99%). Damit ergibt sich für diesen Algorithmus ein theoretischer Speedup von 2. In Tabelle 1 sind die Ausführungszeiten des sequentiellen den Ausführungszeiten des parallelen Minimum-Maximum-Algorithmus für verschieden große Datenmengen gegenübergestellt.

Die Ausführungszeiten aller in dieser Arbeit vorgestellten Algorithmen sind mit der CRAY-spezifischen Funktion IRTC gemessen worden. Diese Funktion gibt den augenblicklichen Inhalt des Echtzeit-Uhrregisters der CRAY X-MP/22 an, welches bei jedem CPU-Zyklus um 1 hochgezählt wird [KFA52,85]. Somit konnte die Anzahl der insgesamt benötigten Takte ermittelt werden, die dann in Sekunden umgerechnet wurde.

Aus dem Verhältnis der auf diese Weise ermittelten sequentiellen bzw. parallelen Ausführungszeit ergibt sich für den Minimum-Maximum-Algorithmus der folgende erreichte Speedup:

Tabelle 1. Ausführungszeiten des sequentiellen und parallelen Minimum-Maximum-Algorithmus und tatsächlich erreichter Speedup.

Anzahl Daten N	ohne Multitasking T_1/s	mit Multitasking T_p/s	erreichter Speedup T_1/T_p
4000	0.00174	0.00155	1.12
8000	0.00349	0.00243	1.44
16000	0.00699	0.00429	1.63
32000	0.01398	0.00778	1.80
64000	0.02827	0.01477	1.91
128000	0.05594	0.02876	1.95
256000	0.11277	0.05673	1.98
512000	0.22418	0.11320	1.98

Tabelle 1 und Abb. 11 verdeutlichen, daß ab N = 256000 der theoretische Speedup von 2 fast erreicht wird. Dies kommt daher, weil für große N der Overhead in Bezug auf die Ausführungszeit vernachlässigbar ist, während er für kleine N noch eine gewichtige Rolle spielt.

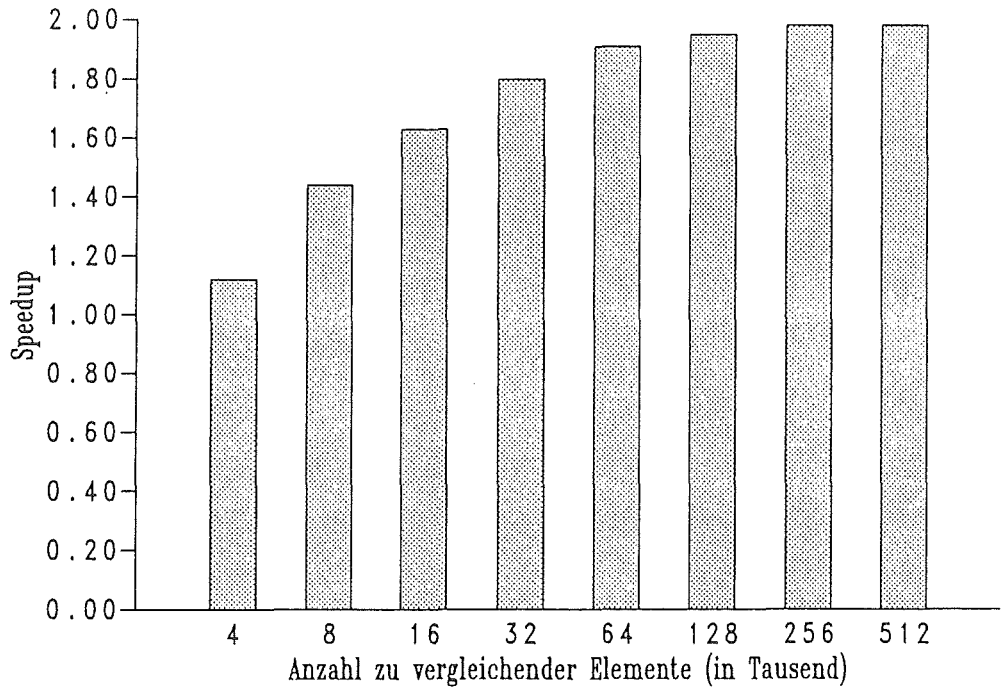


Abb. 11. Speedup-Werte des Minimum-Maximum-Algorithmus als Funktion der Anzahl zu vergleichender Elemente: Das zu durchsuchende Array A wurde mit Zufallszahlen vorbesetzt.

4.1.1.2 Quicksort

Eines der meistbehandelten Probleme der Datenverarbeitung ist das **Sortieren** von Daten. Wegen der Vielzahl vorhandener Sortieralgorithmen steht ein Programmierer, der Daten von einem Rechner sortieren lassen möchte, vor dem Problem, welcher dieser Algorithmen für die Lösung seiner Aufgabe am geeignetsten ist. Aus dieser Vielzahl von Algorithmen hebt sich seiner Leistungsfähigkeit wegen der *Quicksort*-Algorithmus deutlich ab. Untersuchungen haben gezeigt, daß Quicksort der effizienteste universelle Sortieralgorithmus ist [Sedg,78]. Neben der leichten Verständlichkeit ist Quicksort im Vergleich zu seinen schärfsten Konkurrenten wie *Heapsort* und *Mergesort* im Mittel bis zu zweimal schneller [Wirth,79]. Diese Vorausset-

zungen führten dazu, Quicksort im Hinblick auf Parallelisierbarkeit und Beschleunigung durch Multitasking zu untersuchen. Für diese Untersuchungen wurde eine nichtrekursive Version des Quicksort-Algorithmus verwendet.
Die Problemstellung für einen Sortieralgorithmus lautet: Gegeben ist ein Array A von N reellen Zahlen, die nach ausgeführter Sortierung in aufsteigender Reihenfolge $A[1] \leq A[2] \leq \ldots \leq A[N]$ angeordnet sein sollen.
Quicksort ist eine Sortiermethode, die nach dem Prinzip "teile und herrsche" arbeitet und das Array A zunächst so aufteilt, daß folgende Bedingungen gelten:

1. Ein Element x steht an der richtigen Stelle im Array. Ist es das j-kleinste, dann befindet es sich in Position A[j].
2. Alle Elemente, die links von A[j] stehen, sind $\leq A[j]$. Diese Elemente A[1], A[2], ..., A[j-1] bilden das "linke Teilarray".
3. Alle Elemente rechts von A[j] sind $\geq A[j]$. Diese Elemente A[j+1], ..., A[N] bilden das "rechte Teilarray".

Nach dieser ersten Partition wendet man das gleiche Verfahren auf das linke bzw. rechte Teilarray an. Dies wird solange fortgesetzt, bis keine Zerlegung mehr möglich und damit das gesamte Array sortiert ist. Um das wiederholte Partitionieren ausführen zu können, muß eine Liste der notwendigen Zerlegungen geführt werden. Nach jedem Zerlegungsschritt ergibt sich die Forderung nach zwei weiteren Zerlegungen. Da jedoch nur eine in der anschließenden Iteration ausgeführt werden kann, muß die andere in der Liste gespeichert werden. Die Liste wird dabei wie ein "push-down"-Stack aufgebaut. Zur Implementierung der Liste verwendet man eine Array-Variable *stack* und einen Zeiger S, der die letzte Eintragung in der Liste anzeigt. Die linke Grenze eines Teilarrays steht in *stack*$[S,1]$ und die rechte in *stack*$[S,2]$. Die Prozeduren **push** und **pop** schreiben auf bzw. lesen vom Stack. Der sequentielle Quicksort-Algorithmus kann wie folgt beschrieben werden:

Algorithmus *Quicksort*

```
push(1,N)
repeat                    {hole die obersten Grenzen vom Stack}
   pop(ℓ,r)
   repeat                 {zerlege A[ℓ], ..., A[r]}
      wähle Vergleichselement x := A[(ℓ+r) div 2]
      i := ℓ und j := r
      repeat
         while A[i] < x do i := i + 1
         while x < A[j] do j := j - 1
         if i ≤ j then
            tausche A[i] und A[j]
            i := i + 1
            j := j - 1
      until i > j
      if i < r then
         push(i,r)        {lege Grenzen des rechten Teilarrays ab}
      r := j              {zerlege linkes Teilarray weiter}
   until ℓ ≥ r
until S = 0
```

Bei Betrachtung des Algorithmus stellt sich die Frage, was passiert, wenn in einer oder in beiden **while**-Anweisungen Gleichheit gilt. Dann müssen zusätzliche Austausche in Kauf genommen werden, die jedoch im mittleren "zufälligen" Fall relativ selten vorkommen. Würden die Bedingungen dieser Anweisungen auf Gleichheit abgeändert, dann könnte x nicht mehr als Marke dienen. Darüberhinaus würde ein Array mit identischen Schlüsseln bewirken, daß das Durchlaufen über die Grenzen des Array hinausgeht, wenn nicht kompliziertere Abbruchbedingungen verwendet würden [Wirth,79].
Die obige Form des Algorithmus legt immer die Grenzen des rechten Teilarrays ab, während das linke weiter zerlegt wird. Man kann nun die notwendige Länge der Liste dadurch beschränken, daß man die Grenzen des größeren der Teilarrays einer Zerlegung auf dem Stack ablegt, während das kleinere weitersortiert wird. Durch diese Modifikation kann die Länge der Liste nach unten auf logN (logN bezeichne im weiteren $\log_2 N$) beschränkt werden [Wirth,79].

Angenommen man trifft bei der Wahl des Vergleichselements x immer das mittlere der N Elemente. Dann teilt jede Zerlegung das Array in zwei Hälften, und zum Sortieren sind logN Durchläufe notwendig. Mit den N Vergleichen, die dabei ausgeführt werden, ergibt sich eine Zeitkomplexität von O(N*logN). Untersuchungen haben gezeigt, daß dieses Verhalten auch das durchschnittliche Zeitverhalten von Quicksort ist [Wirth,79], [Sedg,83]. Betrachtet man jedoch den ungünstigsten Fall, bei dem jedesmal das größte Element einer Zerlegung als Vergleichselement x gewählt wird, so wird bei jedem Durchlauf ein Segment mit N_s Elementen in einen linken Teil mit N_s-1 und einen rechten Teil mit nur einem Element zerlegt. Dies hat zur Folge, daß statt logN nun N Zerlegungen ausgeführt werden und Quicksort zum "Slowsort" wird. Damit ist das Zeitverhalten im schlimmsten Fall von der Größenordnung $O(N^2)$.
Um das Eintreten des schlimmsten Falls unwahrscheinlich zu machen, bedient man sich der "median-of-three"-Methode. Bei dieser Methode wird das mittlere von drei Elementen, z.B. vom ersten, dem in der Mitte und vom letzten Element, als Vergleichselement x gewählt. Diese Methode schließt zum einen den ungünstigsten Fall für Quicksort mit großer Wahrscheinlichkeit aus, und zum anderen kann mit ihrer Hilfe die durchschnittliche Leistung von Quicksort verbessert werden [Sedg,78].
Weil das Zeitverhalten von Quicksort für kleine Arrays schlecht im Gegensatz zu anderen Sortierverfahren ist, kann eine weitere Leistungsverbesserung dadurch erreicht werden, daß Teilarrays der Länge ≤ M nicht weiter zerlegt werden, sondern mit Hilfe eines anderen Sortierverfahrens sortiert werden. Zum Sortieren kleiner Arrays eignet sich das Verfahren des **direkten Einfügens** [Wirth,79] besonders gut. Untersuchungen haben ergeben, daß der beste Wert für M bei 9 liegt [Sedg,78]. Aus diesen Verbesserungsmöglichkeiten ergibt sich ein etwas veränderter sequentieller Quicksort-Algorithmus, der sich anhand der folgenden Überlegungen parallelisieren läßt.

Nachdem das gesamte Array durch die erste Partition in zwei Teile zerlegt worden ist, können diese beiden disjunkten Teilarrays von zwei Tasks weiterzerlegt und sortiert werden. Dazu ist es erforderlich, daß beide Tasks den Inhalt der Liste kennen, d.h. die Variable *stack* wird als gemeinsame Variable vereinbart. Zur Vermeidung von Konflikten und Fehlern werden die Operationen auf dem Stack in kritische Sektionen eingeschlossen. Nachdem eine Task eine Zerlegung abgeschlossen hat, überprüft

sie, ob auf dem Stack noch Information für eine weitere Zerlegung steht. Ist dies der Fall, wird der nächste Abschnitt, d.h. der nächste Zerlegungsdurchlauf gestartet. Andernfalls wird die Task abgeschlossen.
Der parallele Algorithmus funktioniert nun derart, daß die Haupt-Task eine andere Task erzeugt, die das Array in zwei Teilarrays zerlegt. Nachdem die Beendigung dieser ersten Zerlegung an die Haupt-Task signalisiert wurde, kann diese die Grenzen des einen Teils der ersten Zerlegung vom Stack lesen und anschließend parallel und asynchron zur ersten Task dieses Teilarray weiterzerlegen. Damit ergibt sich aus dem verbesserten sequentiellen der folgende asynchrone parallele Quicksort-Algorithmus. Die Variablen *sleer*, *warten0* und *warten1* sind globale logische Variablen. Die Variable *sleer* zeigt an, ob der Stack leer ist. Die Variable *warten0* zeigt an, daß Task0 entweder warten muß, bis Task1 den leeren Stack neu beschrieben hat, oder daß Task0 das Ausführungsende von Task1 abwartet. Muß Task1 darauf warten, daß Task0 den leeren Stack neu beschreibt, so wird dies durch *warten1* angezeigt. Der Wert der Variablen *warten1* ist ebenfalls wahr, wenn Task1 ihre Ausführung beendet hat. Die lokale Variable *aktiv* in Task0 gibt an, ob Task1 noch tätig ist.

Algorithmus *Quicksort_Parallel*

```
Task0 führt aus:
   sleer := true
   warten0 := false
   push(1,N)
   CALL TSKSTART(TSKID1,Task1)
   CALL EVWAIT(EVENT1)
   CALL EVCLEAR(EVENT1)
10 CALL LOCKON(LOCKS)
   if S > 0 then
      pop(ℓ,r)
      CALL LOCKOFF(LOCKS)
      repeat              {Zerlege A[ℓ], ..., A[r]}
         wähle Vergleichselement x := median_of_three(ℓ,r)
         i := ℓ+1 und j := r
         repeat
            while A[i] < x do i := i + 1
            while x < A[j] do j := i - 1
```

```
            if i ≤ j then
               tausche A[i] und A[j]
               i := i + 1
               j := j - 1
         until i > j
         stelle fest, welches der beiden Teilarrays das kleinere ist:
            if das größere ≤ M then
               sortiere beide mit direktes_Einfügen
            else                {lege die Grenzen g_ℓ, g_r des größeren
               CALL LOCKON(LOCKS)      auf dem stack ab}
               if sleer = true then
                  CALL EVPOST(EVENT2)
                  sleer := false
               push(g_ℓ,g_r)
               CALL LOCKOFF(LOCKS)
            if das kleinere ≤ M then
               sortiere es mit direktes_Einfügen
            else
               zerlege das kleinere weiter
      until ℓ ≥ r
      goto 10
   else
      CALL LOCKOFF(LOCKS)
      sleer := true
      aktiv := TSKTEST(TSKID1)
      if aktiv = true then
         if warten1 = false then
            warten0 := true
            CALL EVWAIT(EV1)
            CALL EVCLEAR(EV1)
            warten0 := false
            goto 10
         else
            warten0 := true
            CALL EVPOST(EV2)
      CALL TSKWAIT(TSKID1)
```

```
Task1 führt aus:
   warten1 := false
10 CALL LOCKON(LOCKS)
   if S > 0 then
      pop(ℓ,r)
      CALL LOCKOFF(LOCKS)
      repeat              {Zerlege A[ℓ], ..., A[r]}
         wähle Vergleichselement x := median_of_three(ℓ,r)
         i := ℓ+1 und j := r
         repeat
            while A[i] < x do i := i + 1
            while x < A[j] do j := i - 1
            if i ≤ j then
               tausche A[i] und A[j]
               i := i + 1
               j := j - 1
         until i > j
         stelle fest, welches der beiden Teilarrays das kleinere ist:
            if das größere ≤ M then
               sortiere beide mit direktes_Einfügen
            else                {lege die Grenzen g_ℓ, g_r des größeren
               CALL LOCKON(LOCKS)      auf dem stack ab}
               if sleer = true then
                  CALL EVPOST(EVENT1)
                  sleer := false
               push(g_ℓ,g_r)
               CALL LOCKOFF(LOCKS)
            if das kleinere ≤ M then
               sortiere es mit direktes_Einfügen
            else
               zerlege das kleinere weiter
      until ℓ ≥ r
      goto 10
   else
      CALL LOCKOFF(LOCKS)
      sleer := true
      if warten0 = false then
         warten1 := true
```

```
            CALL EVWAIT(EVENT2)
            CALL EVCLEAR(EVENT2)
            wartenl := false
            goto 10
         else
            wartenl := true
            CALL EVPOST(EVENT1)
```

Der parallele Algorithmus macht deutlich, warum die "median-of-three"-Methode benötigt wird. Würde nämlich der schlimmste Fall für Quicksort eintreten, dann hätte für ein Segment N_s jede Task ein Element zu bearbeiten, während die Grenzen des N_s-1 Elemente umfassenden Teilarrays jeweils auf den Stack geschrieben werden. Dies aber hätte das häufige Eintreten in die kritischen Sektionen, d.h. einen hohen Overhead und damit eine geringe Beschleunigung zur Folge.

Während der Implementierung des parallelen Algorithmus wurde festgestellt, daß Quicksort nicht völlig asynchron parallelisiert werden kann und deshalb als **semi-asynchroner** paralleler Algorithmus bezeichnet werden muß. Während der Ausführung des parallelen Algorithmus kann es nämlich in Abhängigkeit der gewählten Daten vorkommen, daß auf dem Stack nur noch die Grenzen eines Teilarrays stehen. Angenommen beide Tasks wollen nun diese Grenzen lesen. Durch den exklusiven Zugriff liest eine Task die Grenzen, während die zweite bei Freigabe der kritischen Sektion den Stack leer vorfindet und ihre Ausführung beenden würde. Dies hätte zur Folge, daß der Rest der Sortierung nur noch sequentiell ausgeführt wird. Deshalb wurde der parallele Algorithmus geringfügig in der Weise modifiziert, daß eine Task, die den Stack leer vorfindet, durch EVWAIT solange wartet, bis die andere Task durch EVPOST signalisiert, daß der Stack entweder beschrieben oder die Sortierung beendet worden ist. Der dabei entstehende Overhead kann in Bezug auf die Ausführungszeit des parallelen Algorithmus vernachlässigt werden.

Für die Untersuchungen zum Laufzeitverhalten eines Algorithmus erwies sich ein Hilfsmittel zur dynamischen Analyse eines Multitasking-Programms als besonders nützlich. Dieses **"timer trace"** von J. L. Larson, einem Mitarbeiter der Firma CRAY RESEARCH, unterstützt die Verfolgung des Ausführungsflusses eines Multitasking-Jobs [CRAY,222].

Bei der Implementierung des sequentiellen und parallelen Algorithmus wurde das zu sortierende Array A mit Zufallszahlen zwischen 0 und 1

initialisiert. Weil die Leistung von Quicksort sehr stark von der Verteilung der Zahlen im Array abhängt, wurden für jeden der Algorithmen drei Jobs gestartet, wobei die Zufallszahlenfolge jeweils einen anderen Startwert erhielt. Aus den resultierenden Ausführungszeiten, die keine allzu gravierenden Unterschiede aufwiesen, wurden sowohl für den sequentiellen als auch für den parallelen Algorithmus durchschnittliche Ausführungszeiten ermittelt. Daraus ergab sich der in Tabelle 2 enthaltene tatsächlich erreichte Speedup.

Tabelle 2. Ausführungszeiten des sequentiellen und parallelen Quicksort-Algorithmus und tatsächlich erreichter Speedup.

Anzahl Daten N	ohne Multitasking T_1/s	mit Multitasking T_p/s	erreichter Speedup T_1/T_p
4000	0.04291	0.02654	1.62
8000	0.09116	0.05476	1.66
16000	0.19512	0.11459	1.70
32000	0.41560	0.23938	1.74
64000	0.90755	0.51563	1.76
128000	1.94537	1.09794	1.77
256000	4.08164	2.28042	1.79
512000	8.56561	4.74648	1.80

Hieraus und aus Abb. 12 erkennt man, daß die Beschleunigung für kleine Werte N noch recht gering ist. Warum das so ist, wird aus Tabelle 3 ersichtlich. Sie enthält eine Gegenüberstellung des tatsächlich erreichten und des theoretischen Speedups. Der für die Berechnung des theoretischen Speedups notwendige Multitasking-Teil f wurde so ermittelt, indem die Zeit

für die erste Zerlegung des parallelen Algorithmus prozentual auf die Ausführungszeit des sequentiellen Algorithmus umgerechnet wurde. Tabelle 3 zeigt, daß für kleine Werte N der sequentielle Zeitanteil (1-f) des parallelen Algorithmus noch dominiert, während sein Einfluß mit wachsendem N abnimmt.

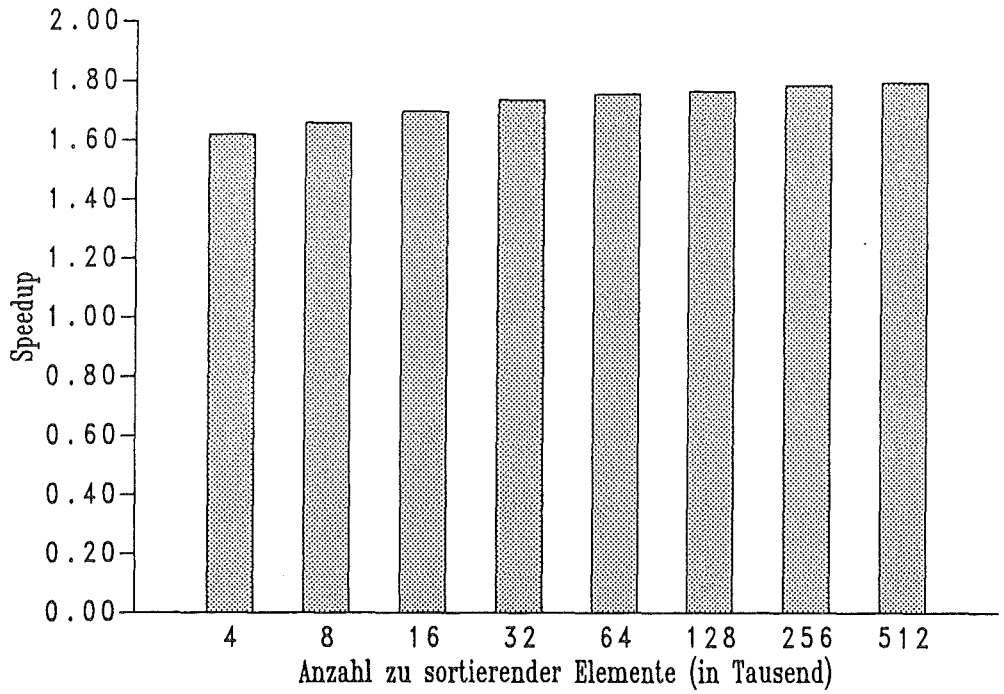


Abb. 12. Erreichte Speedup-Werte des Quicksort-Algorithmus als Funktion der Anzahl zu sortierender Elemente: Das zu sortierende Array A wurde mit Zufallszahlen vorbesetzt.

Weiterhin gibt Tabelle 3 Aufschluß darüber, daß der entstehende Overhead mit wachsendem N zunimmt. Wegen der steigenden Zahl zu sortierender Elemente, wird der Stack häufiger in Anspruch genommen. Dies hat ein häufigeres Eintreten in die kritischen Sektionen und damit einen höheren Overhead zur Folge. Der in Tabelle 3 angegebene Overhead wurde anhand der Gleichung für die parallele Ausführungszeit von Abschnitt 3.1.1 ermittelt. Dieser Overhead setzt sich aus verschiedenen Komponenten zusammen. Zu dem Aufwand, der durch die Verwendung der Multitasking-Routinen entsteht, müssen noch Verzögerungen durch auftretende Speicherbankkonflikte

hinzugezählt werden, wenn beide Tasks überlappt auf zwei verschiedene Elemente des Feldes A zugreifen wollen, die auf derselben Speicherbank abgelegt sind.
Weitere Untersuchungen haben ergeben, daß die Aufrufe der Prozeduren *direktes_Einfügen* und *median_of_three* im Mittel parallel 20 bis 30 Takte mehr kosten als sequentiell. Dieser Aufwand kann vermieden werden, indem der Code dieser Unterprogramme an den entsprechenden Stellen des Programms eingebaut wird. Dies hat jedoch i. allg. den Nachteil einer äußerst unübersichtlichen Programmstruktur.

Tabelle 3. Zusammenhänge zwischen erreichtem und theoretischem Speedup, Multitasking-Teil f und Overhead OH.

Anzahl Daten N	theoretischer Speedup S_{th}	Overhead OH/ms	tatsächlich erreichter Speedup Sp	Multitasking-Teil f
4000	1.87	3.56	1.62	92.9%
8000	1.88	6.31	1.66	93.7%
16000	1.89	11.37	1.70	94.2%
32000	1.90	20.57	1.74	94.7%
64000	1.91	40.07	1.76	95.2%
128000	1.92	82.46	1.77	95.6%
256000	1.92	157.97	1.79	96.0%
512000	1.93	300.93	1.80	96.2%

Abb. 13 zeigt den tatsächlich erreichten und theoretisch maximal möglichen Speedup in Abhängigkeit vom Multitasking-fähigen Teil f des sequentiellen Algorithmus. Der Unterschied zwischen S_{th} und Sp gibt den prozentualen Overhead-Anteil der parallelen Ausführungszeit wieder.

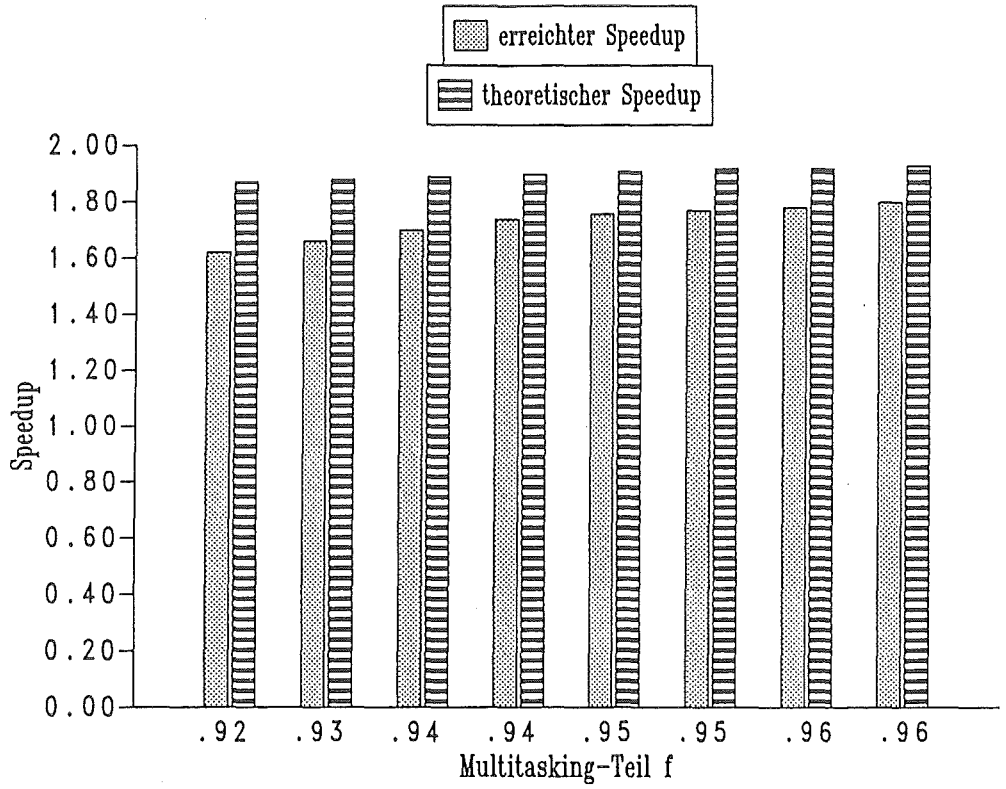


Abb. 13. Theoretische und erreichte Speedup-Werte des Quicksort-Algorithmus als Funktion des Multitasking-Teils f.

Tabelle 4. gibt einen Überblick über die Effizienz und Redundanz des parallelen Quicksort-Algorithmus. Hieraus und aus Abb. 14 wird deutlich, daß mit wachsendem N die Effizienz des Algorithmus zunimmt. Aus der Redundanz erkennt man, daß für große N der Anteil an zusätzlich geleisteter Arbeit prozentual zur insgesamt geleisteten Arbeit abnimmt. Die hohen Werte der Auslastung zeigen, daß der parallele Quicksort-Algorithmus beide Prozessoren äußerst effizient ausnutzt. Zum anderen wird daraus ersichtlich, wie gut die zu leistende Arbeit während der Ausführung dynamisch auf beide Tasks verteilt wird.

Tabelle 4. Effizienz, Redundanz und Auslastung für den parallelen Quicksort-Algorithmus.

Abzahl Daten N	Effizienz E_p	Redundanz R_p	Auslastung U_p
4000	0.810	1.22	0.988
8000	0.830	1.19	0.994
16000	0.850	1.17	0.997
32000	0.870	1.15	0.998
64000	0.880	1.14	0.999
128000	0.885	1.13	0.999
256000	0.895	1.12	0.999
512000	0.900	1.11	0.999

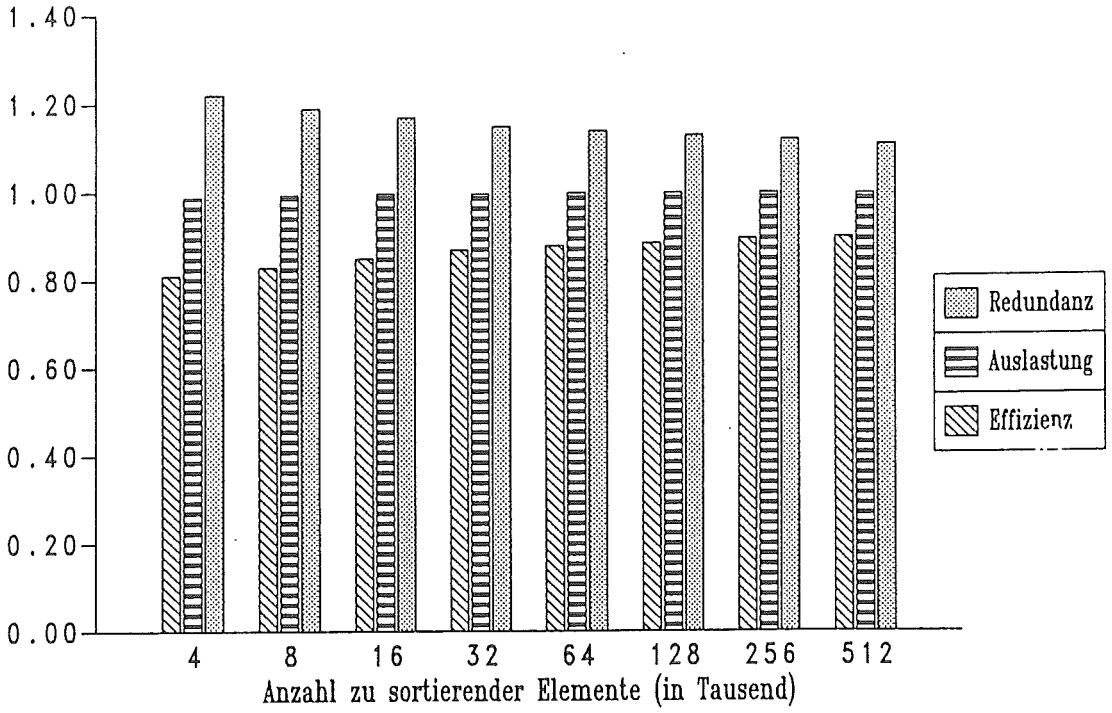


Abb. 14. Effizienz, Redundanz und Auslastung für den Quicksort-Algorithmus als Funktion der Anzahl zu sortierender Elemente

Nach diesem asynchronen bzw. semi-asynchronen parallelen Algorithmus soll nun gezeigt werden, welche Beschleunigung man für synchrone parallele Algorithmen durch Nutzung der Multitasking-Fähigkeiten erreichen kann.

4.1.2 Synchrone parallele Algorithmen

Ein *synchroner* paralleler Algorithmus besitzt eine Ablaufstruktur mit folgender Eigenschaft: Es gibt einen Task-Abschnitt, der solange nicht aktiviert wird, bis eine andere Task das Ende der Ausführung eines bestimmten Abschnitts signalisiert hat. In einem synchronen Algorithmus existieren also zwischen den Abschnitten verschiedener Tasks Präzedenzrelationen, durch die festlegt wird, welche Tasks auf die Ergeb-

nisse anderer Tasks warten müssen. Im folgenden sollen Standardalgorithmen zur Lösung linearer Gleichungssysteme im Hinblick auf ihre Parallelisierbarkeit untersucht werden. Es liegt in der Struktur dieser numerischen Algorithmen, daß sie bei paralleler Ausführung nach jedem Berechnungsabschnitt synchronisieren müssen. Obwohl sich dieser Synchronisationsaufwand nachteilig auf den Speedup auswirken kann, soll hier gezeigt werden, daß durch Ausnutzung der Multitasking-Fähigkeiten für synchrone parallele Algorithmen eine ausreichende Beschleunigung erzielt werden kann.

4.1.2.1 Jacobi-Verfahren

Das *Jacobi-Verfahren* oder *Gesamtschrittverfahren* ist ein Iterationsverfahren zur Lösung linearer Gleichungssysteme. Voraussetzung für die Anwendung des Jacobi-Verfahrens ist, daß das Gleichungssystem

$$Ax = b$$

mit regulärer (n × n) Koeffizientenmatrix A auf eine Fixpunktform

$$x = Tx + u$$

gebracht werden kann. Wenn die Diagonalelemente a_{ii}, i = 1, ..., n, der Matrix A von Null verschieden sind, ist die Matrix $D := (a_{11}, \ldots, a_{nn})$ regulär. Damit läßt sich A in die Form A = L + D + U zerlegen, wobei L eine untere Dreiecksmatrix und U eine obere Dreiecksmatrix ist. Damit ist das Gleichungssystem

$$Ax = b, \quad A = L + D + U, \quad D \text{ regulär, äquivalent zu}$$
$$Dx = -(L+U)x + b \text{ oder } x = -D^{-1}(L+U)x + D^{-1}b.$$

Setzt man nun

$$T := -D^{-1}(L+U) \text{ und } u := D^{-1}b$$

so hat man eine zum Ausgangssystem $Ax = b$ äquivalente Fixpunktform $x = Tx + b$ gefunden. Die Matrix T bezeichnet man als *Jacobi-Matrix* und das Iterationsverfahren hat dann folgende Gestalt [Num,79]:

Sei $x^{(0)}$ ein beliebiger Startvektor:

$$x_i^{(k+1)} = 1/a_{ii}\Big(- \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} + b_i \Big), \quad i = 1, \ldots, n \text{ und } k = 0, 1, 2, \ldots$$

Hinreichende Konvergenzkriterien für das Jacobi-Iterationsverfahren sind:

1. *Spaltensummenkriterium*

$$|a_{ii}| > \sum_{\substack{k=1 \\ k \neq i}}^{n} |a_{ki}| \quad \text{mit } i = 1, \ldots, n.$$

2. *Zeilensummenkriterium*

$$|a_{ii}| > \sum_{\substack{k=1 \\ k \neq i}}^{n} |a_{ik}| \quad \text{mit } i = 1, \ldots, n.$$

Der sequentielle Algorithmus zur iterativen Lösung eines Gleichungssystems $Ax = b$ mit dem Jacobi-Verfahren hat dann folgende Ablaufstruktur:

Algorithmus *Jacobi_Verfahren*

```
o.B.d.A. setze Startvektor x^(0) := 0
for i := 1 to n do                    {wenn a_ii = 0, dann
   if a_ii = 0 then                    vertausche Zeilen von A}
      for j := i+1 to n do
         if a_ji ≠ 0 then
            tausche Zeile_i und Zeile_j
            tausche b_i und b_j
Überprüfung des Zeilen- und Spaltensummenkriteriums
if ein Konvergenzkriterium erfüllt then
```

```
k := 0
repeat
   for i := 1 to n do
```

$$x_i^{(k+1)} := 1/a_{ii}\Big(- \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} + b_i \Big)$$

```
   for i := 1 to n do
      Abbr[i] := x_i^(k+1) - x_i^(k)
   maximum :=  max (|Abbr[1]|, ..., |Abbr[n]|)
   k := k + 1
until maximum ≤ Eps
```

Aus der Iterationsvorschrift des sequentiellen Algorithmus erkennt man, daß die Berechnungen aller x_i, $i = 1, \ldots, n$, in einer Iteration voneinander unabhängig sind. Somit bestehen zwischen den Berechnungen der x_i keine Präzedenzrelationen, und die Arbeit jeder Iteration kann auf zwei Tasks verteilt werden. Nachdem die Matrix A von Task0 im Hinblick auf nichtverschwindende Diagonalelemente untersucht wurde und gegebenenfalls Zeilenvertauschungen vorgenommen worden sind, startet Task0 eine Task1. Jede der beiden Tasks überprüft dann, ob eines der Konvergenzkriterien erfüllt ist. Wenn ja, können beide nach abgeschlossener Synchronisation mit der Ausführung der ersten Iteration beginnen. Nach Bestimmung des ersten Iterationsvektors $x^{(1)}$ kann die Differenzenbildung des Abbruchkriteriums parallel ausgeführt werden. Nachdem jede Task ein lokales Maximum bestimmt hat, kann in einer kritischen Sektion das absolut größte Element der Differenzenbildung gefunden werden. Anschließend müssen beide Tasks synchronisieren. Dieses Element *maximum* ist in beiden Tasks als globale Variable bekannt. Nach Überprüfung des Abbruchkriteriums fahren beide Tasks parallel mit der nächsten Iteration fort. Damit kann der folgende synchrone parallele Algorithmus für das Jacobi-Iterationsverfahren formuliert werden.

Algorithmus *Jacobi_Parallel*

Task0 führt aus:

setze Startvektor $x^{(0)} := 0$

for i := 1 **to** n **do** {wenn $a_{ii} = 0$, dann

if $a_{ii} = 0$ **then** vertausche Zeilen von A}

for j := i+1 **to** n **do**

if $a_{ji} \neq 0$ **then**

tausche $Zeile_i$ und $Zeile_j$

tausche b_i und b_j

CALL TSKSTART(TSKID1,Task1)

Überprüfung des Zeilensummenkriteriums

CALL EVWAIT(EVENT2)

CALL EVPOST(EVENT1)

CALL EVCLEAR(EVENT2)

if kein Kriterium erfüllt **then**

CALL TSKWAIT(TSKID1)

else

iu := n **div** 2 + 1

k := 0

repeat

for i := 1 **to** iu **do**

$$x_i^{(k+1)} := 1/a_{ii}\Big(- \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j^{(k)} + b_i\Big)$$

for i := 1 **to** iu **do**

Abbr[i] := $x_i^{(k+1)} - x_i^{(k)}$

lmax := max (|Abbr[1]|, ..., |Abbr[iu]|)

CALL LOCKON(LOCKM)

if lmax > *maximum* **then**

maximum := lmax

CALL LOCKOFF(LOCKM)

CALL EVWAIT(EVENT2)

CALL EVPOST(EVENT1)

CALL EVCLEAR(EVENT2)

```
        k := k + 1
      until maximum ≤ Eps
      CALL TSKWAIT(TSKID1)
```

Task1 führt aus:

```
   Überprüfung des Spaltensummenkriteriums
   CALL EVPOST(EVENT2)
   CALL EVWAIT(EVENT1)
   CALL EVCLEAR(EVENT1)
   if kein Kriterium erfüllt then
      return
   else
      il := n div 2 + 2
      k := 0
      repeat
         for i := il to n do
```

$$x_i^{(k+1)} := 1/a_{ii}\Big(- \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j^{(k)} + b_i\Big)$$

```
         for i := il to n do
            Abbr[i] := x_i^(k+1) - x_i^(k)
         lmax := max (|Abbr[il]|, ..., |Abbr[n]|)
         CALL LOCKON(LOCKM)
         if lmax > maximum then
            maximum := lmax
         CALL LOCKOFF(LOCKM)
         CALL EVPOST(EVENT2)
         CALL EVWAIT(EVENT1)
         CALL EVCLEAR(EVENT1)
         k := k + 1
      until maximum ≤ Eps
```

Für die Implementierung beider Algorithmen wurde die Matrix A mit Zufallszahlen zwischen 0 und 10 und der Vektor b mit Zufallszahlen zwischen 0 und 100 initialisiert. Die Genauigkeit für die Bestimmung des Lösungsvektors x betrug in beiden Algorithmen Eps = $1*10^{-9}$. Um diese Genauigkeit bei der Bestimmung des Lösungsvektors x zu erreichen, wurden im Mittel für N = 50

ca. 8 Iterationen und für N = 601 etwa 38 Iterationen benötigt. Die durch Rundungsfehler entstehende Ungenauigkeit lag im Mittel für N = 601 bei $5*10^{-6}$.

Voraussetzung für die Anwendung des Jacobi-Iterationsverfahrens auf das Gleichungssystem Ax = b ist, daß eines der Konvergenzkriterien erfüllt sein muß. Deshalb wurden die Diagonalelemente der Matrix A so vorbesetzt, daß diese Voraussetzung erfüllt ist.

Um festzustellen, wie groß der Multitasking-Teil f für diesen Algorithmus im Mittel ist, wurde untersucht, wieviel Zeit daß Suchen nach verschwindenden Diagonalelementen und das Vertauschen von Zeilen in Anspruch nimmt. Dazu wurde die Matrix A nach der Initialisierung so verändert, daß die gesamte erste Spalte durchlaufen und anschließend die erste und letzte Zeile vertauscht werden mußten. Zeitmessungen haben ergeben, daß dieser Zeitanteil in Bezug auf die Gesamtausführungszeit vernachlässigbar ist. Für den Multitasking-Teil f bedeutet dies, daß er beispielsweise für N = 50 98% und für N = 601 99.8% beträgt. Für den theoretischen Speedup ergibt dies Werte von 1.96 bzw. 1.99.

Untersuchungen des parallelen Algorithmus haben gezeigt, daß es wesentlich günstiger ist, die Anzahl der Iterationen für die Berechnung der $x_i^{(k+1)}$ etwas unbalanciert auf beide Tasks zu verteilen. Task1 führt dann eine Iteration weniger aus als Task0. Dies hat zur Folge, daß Task1 in fast jeder Iteration der **repeat**-Schleife ihre kritische Sektion schon verlassen hat, wenn Task0 mit der Ausführung ihres Eintritt-Protokolls beginnt. Daraus ergibt sich zusätzlich, daß CALL EVPOST(EVENT2) vor CALL EVWAIT(EVENT2) ausgeführt wird. Mit dieser Änderung der Verteilung konnten beispielsweise bei N = 601 pro Iteration der **repeat**-Schleife in jeder Task ca. 20% an Overhead eingespart werden.

Tabelle 5 gibt Aufschluß über den tatsächlich erreichten Speedup des Jacobi-Iterationsverfahrens.

Tabelle 5. Ausführungszeiten und erreichter Speedup des Jacobi-Iterationsverfahrens bei Lösung des linearen Gleichungssystems Ax = b.

Größe der Matrix A N × N	ohne Multitasking T_1/s	mit Multitasking T_p/s	erreichter Speedup T_1/T_p
50	0.00299	0.00248	1.21
101	0.00907	0.00584	1.55
201	0.02676	0.01625	1.65
301	0.06777	0.04094	1.66
401	0.12272	0.07341	1.67
501	0.19261	0.10716	1.80
601	0.28686	0.15684	1.83

Abb. 15 zeigt eine Gegenüberstellung des theoretischen und des erreichten Speedups. Der Unterschied zwischen theoretischem und erreichtem Speedup macht deutlich, daß bei kleiner Dimension des Gleichungssystems der Aufwand für die Ausführung der Bibliotheksroutinen zu hoch ist.

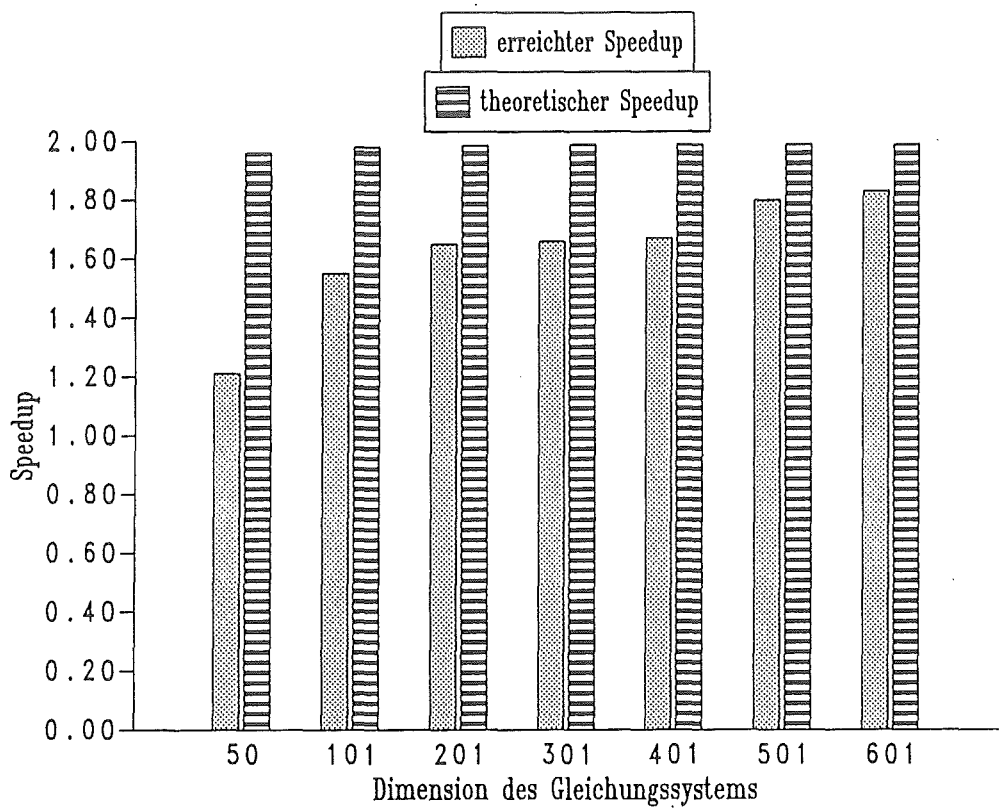


Abb. 15. Theoretische und erreichte Speedup-Werte des Jacobi-Iterationsverfahrens: variable Dimension des Gleichungssystems $Ax = b$; die Elemente der Matrix A und des Vektors b wurden durch Zufallszahlen erzeugt.

Tabelle 6 macht deutlich, warum der Speedup in Abb. 15 von N = 201 bis N = 401 stagniert. Untersuchungen der Granularität für die Berechnung der $x_i^{(k+1)}$ des sequentiellen und parallelen Algorithmus haben ergeben, daß die parallelen Tasks wegen auftretender Speicherbankkonflikte bei der Berechnung der $x_i^{(k+1)}$ verzögert werden. Tabelle 6 ist zu entnehmen, daß diese Verzögerungen bis N = 301 zunehmen und dann bis einschließlich N = 401 in Bezug auf die sequentielle Granularität konstant bleiben. Der Anstieg des Speedups von N = 401 nach N = 501 kommt dadurch zustande, daß der CFT-Compiler ab N = 430 die Berechnung von $a_{ij}x_j^{(k)}$ ($j = 1, \ldots, n$ und $j \neq i$) für jedes i durch die Vektorfunktion SDOT ersetzt und damit die Speicherbankkonflikte nicht mehr so gravierend sind.

Tabelle 6. Durchschnittliche Granularität für die sequentielle bzw. parallele Berechnung der $x_i^{(k+1)}$ und prozentualer Verzögerungsanteil in Bezug auf die sequentielle Granularität.

Größe der Matrix A N × N	Granularität sequentiell in CPU-Zyklen	Granularität parallel je Task in CPU-Zyklen	Anteil der Verzögerung je Task
50	26500	14100	3.2%
101	69000	37000	3.6%
201	174500	96000	5.0%
301	360000	210000	8.3%
401	516000	300500	8.23%
501	646000	345000	3.4%
601	720000	385000	3.4%

Das Jacobi-Iterationsverfahren hat jedoch gegenüber anderen Verfahren zur Lösung linearer Gleichungssysteme den Nachteil, daß es nur dann die Lösung von Ax = b liefert, wenn die Matrix A so beschaffen ist, daß sie einem der oben genannten Konvergenzkriterien genügt.
Läßt sich das Gleichungssystem Ax = b nicht mit Hilfe des Jacobi-Iterationsverfahrens lösen, so kann man zu dessen Lösung **direkte Verfahren** heranziehen.

4.1.2.2 LU-Faktorisierung

Die direkten Verfahren zur Lösung linearer Gleichungssysteme beruhen auf der **Reduktion** der Matrix A auf Dreiecksgestalt mit anschließender *Rücksubstitution* zur Berechnung der Unbekannten.
Eines dieser Reduktionsverfahren ist die *LU-Fatorisierung*. Sie zerlegt die nichtsinguläre (n × n) Matrix A des Gleichungssystems Ax = b in zwei Dreiecksmatrizen

$$A = LU,$$

wobei L eine untere Dreiecksmatrix und U eine *normierte* obere Dreiecksmatrix ist. Damit kann das Gleichungssystem geschrieben werden als

$$LUx = b.$$

Setzt man nun

$$Ly = b \text{ und}$$
$$Ux = y,$$

so erhält man nach Faktorisierung der Matrix A den Vektor y durch *Vorwärtseinsetzen* (erst y_1 berechnen, dann y_2 usw.) und danach den Lösungsvektor x durch Rücksubstitution (erst x_n berechnen, dann x_{n-1} usw.). Zur Gewährleistung der numerischen Stabiblität des Algorithmus wird für jeden Berechnungsdurchlauf eine Teilpivotisierung durchgeführt. Man wählt dabei das betragsgrößte Element jeder Spalte als Pivotelement und vertauscht die entsprechenden Zeilen. Dadurch erhält man nach durchgeführter Faktorisierung eine Permutationsmatrix PA der Matrix A, so daß PA = LU gilt.
Der sequentielle Algorithmus der LU-Faktorisierung mit Teilpivotisierung kann folgendermaßen formuliert werden:

Algorithmus *LU_Faktorisierung*

```
for k := 1 to n-1 do
   finde p so daß
      |a_pk| := max (|a_kk|, ..., |a_nk|)
   ipvt[k] := p                                  {Pivotzeile}
   if p ≠ k then                                 } T_k^k
      for j := 1 to n do
         tausche a_pj und a_kj
   c := 1/a_kk
   for j := k+1 to n do
      a_kj := a_kj*c                             {U-Elemente}
   for j := k+1 to n do                          } T_k^j
      for i := k+1 to n do
         a_ij := a_ij - a_ik*a_kj                {L- und U-Elemente}
```

Ist die Matrix A mit Hilfe der LU-Faktorisierung zerlegt worden, läßt sich der sequentielle Algorithmus zur Lösung des Gleichungssystems Ax = b wie folgt formulieren [DoEi,84]:

Algorithmus *Lösung_Gls_LU*

```
for k := 1 to n do
   x_k := b_k
for k := 1 to n do
   p := ipvt[k]
   tausche x_p und x_k
for k := 1 to n do                               {Lösung von Ly = b}
   t := x_k/a_kk
   for i := k+1 to n do
      x_i := x_i - a_ik*t
   x_k := t
for k := n downto 1 do                           {Lösung von Ux = y}
   t := x_k
   for i := 1 to k-1 do
      x_i := x_i - a_ik*t
```

Ausgehend vom sequentiellen Algorithmus haben Lord, Kowalik und Kumar [LKK,83] die parallele Struktur der LU-Faktorisierung bez. der Folge der Berechnungen untersucht. Definiert man die Berechnungsfolge wie folgt:

$$\{ T_k^j \mid 1 \le k \le j \le n,\ k \le n-1 \},$$

so stellt man fest, daß zwischen den Berechnungen

$$\{ T_k^{k+1}, T_k^{k+2}, \ldots, T_k^n \}$$

keine Präzedenzrelationen bestehen und diese somit parallel ausgeführt werden können. Aus der vollständigen Analyse der Berechnungsfolge ergibt sich der in Abb. 16 dargestellte Präzedenzgraph der LU-Faktorisierung. Die Folge $T_k^{k+1}, \ldots, T_k^n$ kann somit halbiert und auf zwei Tasks aufgeteilt werden.

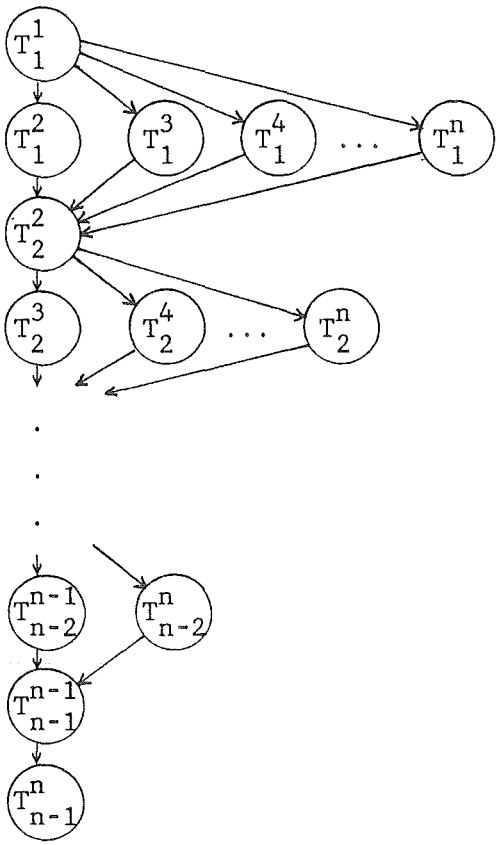


Abb. 16. Präzedenzgraph der LU-Faktorisierung

Der parallele Algorithmus der LU-Faktorisierung mit Teilpivotisierung hat dann folgende Gestalt:

Algorithmus *LU_Faktorisierung_Parallel*

```
Task0 führt aus:
   CALL TSKSTART(TSKID1,Task1)
   for k := 1 to n-1 do
      finde p, so daß
         |a_pk| := max (|a_kk|, ..., |a_nk|)
      ipvt[k] := p                              {Pivotzeile}
      if p ≠ k then
         for j := 1 to n do
            tausche a_pj und a_kj
      CALL EVPOST(EVENT1)
      iu := (n+k+1) div 2 + 1
      if k = n-1 then
         iu := iu - 1
      c := 1/a_kk
      for j := k+1 to iu do
         a_kj := a_kj*c                         {U-Elemente}
      for j := k+1 to iu do
         for i := k+1 to n do
            a_ij := a_ij - a_ik*a_kj            {L- und U-Elemente}
      CALL EVWAIT(EVENT2)
      CALL EVCLEAR(EVENT2)
   CALL TSKWAIT(TSKID1)

Task1 führt aus:
   for k := 1 to n-1 do
      il := (n+k+1) div 2 + 2
      CALL EVWAIT(EVENT1)
      CALL EVCLEAR(EVENT1)
      c := 1/a_kk
      for j := il to n do
         a_kj := a_kj*c                         {U-Elemente}
      for j := il to n do
```

```
for i := k+1 to n do
    a_ij := a_ij - a_ik*a_kj                    {L- und U-Elemente}
CALL EVPOST(EVENT2)
```

Bei der Implementierung der sequentiellen bzw. parallelen LU-Faktorisierung ist darauf zu achten, daß nach jedem Vertauschen von Zeilen abgefragt werden muß, ob $a_{kk} \neq 0$ ist. Weiterhin muß nach den n-1 Iterationen überprüft werden, ob $a_{nn} \neq 0$ ist. Sind diese Bedingungen nicht erfüllt, wird die Ausführung abgebrochen. Wenn $a_{nn} \neq 0$ ist, muß ipvt[n] = n gesetzt werden, da sonst der Algorithmus zur anschließenden Bestimmung des Lösungsvektors x nicht korrekt arbeitet.

Für die Implementierung der sequentiellen und parallelen LU-Faktorisierung wurden die Matrix A und der Vektor b mit Zufallszahlen zwischen 0 und 100 initialisiert. Untersuchungen haben gezeigt, daß das Erzeugen der Zufallszahlenfolge mit verschiedenen Startwerten nur geringe Auswirkungen auf die Ausführungszeiten beider Algorithmen hat. Matrizen mit verschiedenen Daten wirken sich nur auf den Teil der Ausführungszeit aus, der zur Pivotisierung und zum Vertauschen von Zeilen benötigt wird. In Bezug auf die Gesamtausführungszeit des sequentiellen Algorithmus betrug dieser Anteil beispielsweise bei N = 300 und Matrizen mit verschiedenen Daten zwischen 2.8% und 3.0%. Für das Pivotisieren wurde dabei eine Vektorfunktion ISAMAX benutzt, mit deren Hilfe dieser Zeitanteil erheblich reduziert werden konnte. Wichtigkeit erlangt die Zeit für das Pivotisieren und Vertauschen bei der Ermittlung des Multitasking-Teils f, der zur Bestimmung des theoretischen Speedups benötigt wird. Wie man aus der parallelen LU-Faktorisierung erkennt, ist dies jener Teil des Algorithmus, der nicht parallel ausgeführt werden kann. Dazu kommt noch die Zeit, die für das Lösen der beiden Gleichungssysteme Ly = b und Ux = y benötigt wird. Daraus ergab sich beispielsweise im Mittel für N = 300 ein Multitasking-Teil von 95.9% und für N = 600 ein Multitasking-Teil von 97.5%.

Bei den Zeitmessungen für beide Algorithmen wurde festgestellt, daß die Ausführungszeiten sehr stark davon abhängen, ob man die Matrix A als Parameter übergibt oder sie als COMMON-Variable vereinbart. Die besten Ausführungszeiten ergeben sich dann, wenn die Matrix A im sequentiellen Algorithmus als COMMON-Variable vereinbart und im parallelen Algorithmus als Parameter übergeben wird.

Untersuchungen des während der Ausführung entstehenden Overheads haben gezeigt, daß die Synchronisation am Interaktionspunkt von EVENT1 für N =

600 durchschnittlich etwa 4200 Takte kostete. Die Synchronisation am Interaktionspunkt von EVENT2 hatte im Mittel einen Overhead von 800 Takten zur Folge. Der große Unterschied kommt dadurch zustande, weil Task1 in jeder k-ten Iteration warten muß, bis Task0 die Pivotisierung und das Vertauschen von Zeilen abgeschlossen hat. Dadurch ist der Overhead am Interaktionspunkt von EVENT1 wesentlich höher als bei EVENT2. Durch die etwas unbalancierte Verteilung der Arbeit wurde erreicht, daß Task1 CALL EVPOST(EVENT2) in jeder Iteration schon ausgeführt hat, wenn Task0 CALL EVWAIT(EVENT2) ausführt. Wäre die Arbeit gleichmäßig auf beide Tasks verteilt worden, dann hätte der Overhead bei EVENT1 zwar 700 bis 800 Takte weniger gekostet, jedoch für EVENT2 etwa 2000 Takte mehr betragen. Obwohl sich dieser Gewinn für N = 600 nicht sehr stark auf den Speedup auswirkte, konnten durch die etwas ungleichmäßige Verteilung der Arbeit pro Iteration im Mittel etwa 1200 Takte gespart werden. Aus diesen Untersuchungen ergab sich für N = 600 für jede k-te Iteration durch die Verwendung der EVENT-Routinen ein Overhead von durchschnittlich insgesamt 5000 Takten. Für den in jeder k-ten Iteration insgesamt entstehenden Overhead müssen noch Verzögerungen infolge auftretender Bankkonflikte berücksichtigt werden.

Tabelle 7 zeigt die Ausführungszeiten der sequentiellen bzw. parallelen LU-Faktorisierung einschließlich der sequentiellen Lösung des Gleichungssystems $Ax = b$. Daneben gibt sie Aufschluß über den durch Nutzung der Multitasking-Fähigkeiten erreichten Speedup und stellt diesem den theoretischen Speedup gegenüber. Aus dem Unterschied zwischen theoretischem und erreichtem Speedup erkennt man, daß bei kleiner Dimension des Gleichungssystems der Synchronisationsaufwand zu hoch ist.

Tabelle 7. Sequentielle und parallele Ausführungszeiten der LU-Faktorisierung einschließlich der Lösung des Gleichungssystems Ax = b.

Größe der Matrix A N × N	ohne Multitasking T_1/s	mit Multitasking T_p/s	theoretischer Speedup S_{th}	erreichter Speedup T_1/T_p
50	0.00277	0.00327	1.61	0.85
100	0.01347	0.01033	1.80	1.30
200	0.07547	0.04684	1.87	1.61
300	0.21463	0.12458	1.92	1.72
400	0.47901	0.28089	1.90	1.71
500	0.88513	0.48801	1.95	1.81
600	1.48592	0.81935	1.95	1.81

Abb. 17 zeigt den Verlauf des tatsächlich erreichten und des theoretischen Speedups. Man erkennt, daß sowohl der theoretische als auch der erreichte Speedup nicht kontinuierlich ansteigen, sondern zwischen N = 300 und N = 400 etwas abfallen. Ursache dafür ist, daß bei N = 400 die für das Pivotisieren und Vertauschen von Zeilen insgesamt benötigte Zeit in Relation zur Gesamtausführungszeit höher ist als bei N = 300. Dies hat einen geringeren Multitasking-Teil und damit einen Abfall des theoretischen bzw. des erreichten Speedups zur Folge.

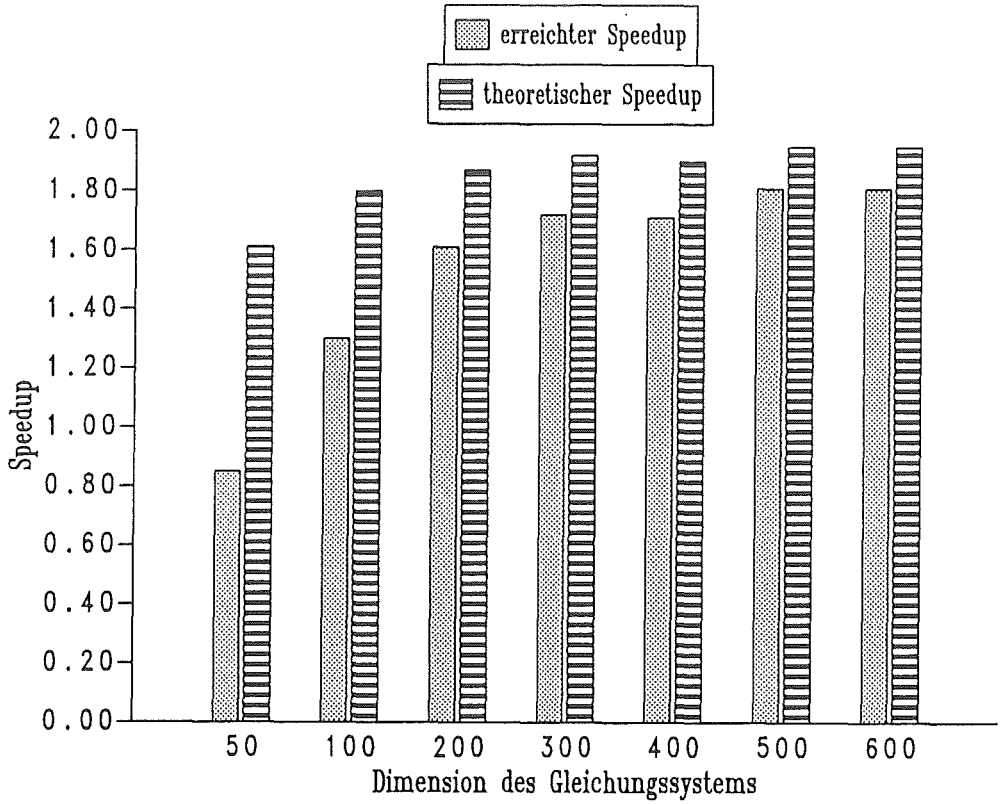


Abb. 17. Theoretische und erreichte Speedup-Werte der LU-Faktorisierung einschl. Lösung des Gleichungsystems Ax = b: variable Dimension des Gleichungssystems; die Elemente der Matrix A und des Vektors b wurden durch Zufallszahlen erzeugt.

Es stellt sich nun die Frage der Parallelisierung des Algorithmus zur Lösung des Gleichungssystems Ax = b bei gegebener LU-Faktorisierung der Matrix A. Bei Betrachtung des Algorithmus erkennt man, daß sowohl die Berechnung der y_i bei der Lösung von Ly = b, als auch die Berechnung der x_i bei der Lösung von Ux = y parallel ausgeführt werden kann. Aus Tabelle 8 erkennt man, daß die durchschnittliche Granularität für die Berechnung der y_i und x_i äußerst gering ist. Weil nach jeder k-ten Iteration synchronisiert werden müßte, hätte dies pro Iteration einen Overhead von mindestens 3000 Takten zur Folge. Daraus ersieht man, daß durch die parallele Ausführung von Ly = b bzw. Ux = y keine Beschleunigung erzielt werden kann, obwohl der theoretische Speedup für beide bei 1.99 liegt. Im Gegen-

teil, eine Ausführung unter Nutzung der Multitasking-Fähigkeiten würde hier eine Verzögerung bewirken.

Tabelle 8. Granularität für die Lösung der Gleichungssysteme Ly = b und Ux = y sowie durchschnittliche Granularität für die Berechnung der y_i und x_i.

Größe der Matrix A N × N	Granularität für Lösung von Ly = b in CPU-Zyklen	Granularität für Lösung von Ux = y in CPU-Zyklen	Granularität für y_i und x_i in CPU-Zyklen
50	10178	8157	156
100	24321	20153	196
200	63820	55818	270
300	119051	106692	348
400	190454	174182	426
500	275979	254598	504
600	378459	352389	580

4.1.2.3 Cholesky-Dekomposition

Ein anderes Verfahren zur Reduktion der Matrix eines linearen Gleichungssystems auf Dreiecksgestalt ist die *Cholesky-Dekomposition*. Voraussetzung für die Anwendung dieses Verfahrens ist, daß die (n × n) Matrix A des Gleichungssystems Ax = b *symmetrisch* und *positiv definit* ist [Stew,73]. Die Cholesky-Dekomposition reduziert die Matrix A auf eine untere Dreiecksmatrix L, so daß gilt:

$$A = LL^t,$$

wobei L^t die Transponierte der unteren Dreiecksmatrix L ist. Damit kann Ax = b geschrieben werden als

$$LL^t x = b \text{ und mit}$$
$$Ly = b$$
$$L^t x = y$$

erhält man den Vektor y wiederum durch Vorwärtseinsetzen und den Lösungsvektor x durch Rücksubstitution.
Der sequentielle Algorithmus zur Reduktion der symmetrischen, positiv definiten Matrix A nach dem Verfahren von Cholesky kann folgendermaßen formuliert werden [DoEi,84]:

Algorithmus *Cholesky_Reduktion*

```
for k := 1 to n do
   for i := 1 to k-1 do
T_k^i |    a_kk := a_kk - a_ki*a_ki
      |    for j := k+1 to n do
      |       a_jk := a_jk - a_ki*a_ji
      |
      | a_kk := √a_kk
T_k^k | c := 1/a_kk
      | for j := k+1 to n do
      |    a_jk := a_jk*c                {Forme k-te Spalte von L}
```

Nachdem die Matrix A auf die untere Dreiecksmatrix L reduziert wurde, kann das Gleichungssystem Ax = b mit Hilfe des folgenden Algorithmus gelöst werden [DoEi,84].

Algorithmus *Lösung_GLS_Cholesky*

```
for k := 1 to n do
   x_k := b_k
for k := 1 to n do                    {Lösung von Ly = b}
   t := x_k/a_kk
   for i := k+1 to n do
      x_i := x_i - a_ik*t
   x_k := t
for k := n downto 1 do                {Lösung von L^t x = y}
   t := x_k/a_kk
   for i := 1 to k-1 do
      x_i := x_i - a_ki*t
   x_k := t
```

Die Folge der Berechnungen in der sequentiellen Cholesky-Reduktion läßt sich wie folgt definieren:

$$\{ T_k^i \mid 1 \leq i < k \leq n,\ i \leq n-1 \},$$

Man stellt dabei fest, daß zwischen den Berechnungen

$$\{ T_k^1, T_k^2, \ldots, T_k^{k-1} \}$$

keine Präzedenzrelationen bestehen und diese somit parallel ausgeführt werden können. Abb. 18 zeigt den Präzedenzgraphen dieser Berechnungsfolge.

Für die Parallelisierung der Cholesky-Reduktion heißt das, daß die Folge $T_k^1, \ldots, T_k^{k-1}$ auf zwei Tasks verteilt und parallel ausgeführt werden kann. Nach Beendigung dieser Berechnungsfolge müssen beide Tasks kommunizieren und synchronisieren. Der parallelen Algorithmus der Cholesky-Reduktion läßt sich dann etwa wie folgt formulieren:

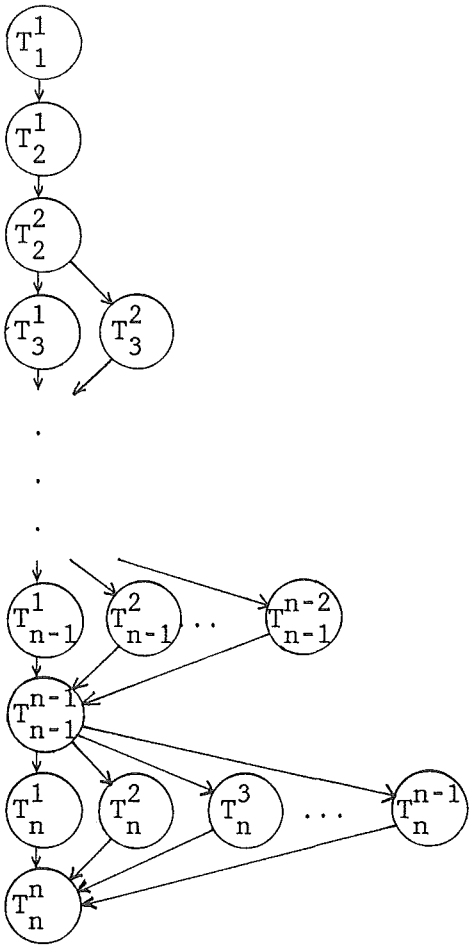


Abb. 18. Präzedenzgraph der Cholesky-Reduktion

Algorithmus *Cholesky_Reduktion_Parallel*

```
Task0 führt aus:
   CALL TSKSTART(TSKID1,Task1)
   for k := 1 to n do
      iu := (k-1) div 2 + 1
      if k = 1 then
         iu := 0
      for i := 1 to iu do
         a_kk := a_kk - a_ki*a_ki
         for j := k+1 to n do
            a_jk := a_jk - a_ki*a_ji
      CALL EVWAIT(EVENT2)
```

```
        CALL EVCLEAR(EVENT2)
        a_kk := √a_kk
        c := 1/a_kk
        for j := k+1 to n do
            a_jk := a_jk*c                    {Forme k-te Spalte von L}
        CALL EVPOST(EVENT1)
    CALL TSKWAIT(TSKID1)
```

Task1 führt aus:

```
    for k := 1 to n do
        il := (k-1) div 2 + 2
        for i := il to n do
            a_kk := a_kk - a_ki*a_ki
            for j := k+1 to n do
                a_jk := a_jk - a_ki*a_ji
        CALL EVPOST(EVENT2)
        CALL EVWAIT(EVENT1)
        CALL EVCLEAR(EVENT1)
```

Für die Implementierung wurde eine Matrix C mit Zufallszahlen zwischen 0 und 10 initialisiert. Um eine symmetrische und positiv definite Matrix A zu generieren, wurde die Matrix C mit der transponierten Matrix C^t multipliziert. Der Vektor b wurde ebenfalls mit Zufallszahlen zwischen 0 und 100 vorbesetzt. Untersuchungen haben ergeben, daß auftretende Ungenauigkeiten bei der Bestimmung des Lösungsvektors x von der Verteilung der Zufallszahlen abhängen. Während diese Ungenauigkeiten im überwiegenden Teil der Fälle unter $1*10^{-5}$ lagen, betrugen sie bei N = 500 maximal $1*10^{-3}$.

Bei der Implementierung der sequentiellen bzw. parallelen Cholesky-Reduktion ist darauf zu achten, daß nach jedem Berechnungsabschnitt T_k^i untersucht wird, ob das führende Element a_{kk} der k-ten Untermatrix ≤ 0 ist. Ist dies der Fall, muß die Ausführung abgebrochen werden. Bei der Implementierung des parallelen Algorithmus ist zusätzlich darauf zu achten, daß die a_{kk} und a_{jk} jeder Berechnungsfolge nicht von beiden Tasks gleichzeitig verändert werden. Die Lösung dieses Problems mittels kritischer Sektionen wäre wenig sinnvoll, weil dadurch die Arbeit der Berechnungsfolge $T_k^1, \ldots, T_k^{k-1}$ nicht mehr parallel ausgeführt werden könnte. Das Problem läßt sich beispielsweise dadurch lösen, daß man für Task1 statt

der a_{kk} und a_{jk} ein globales, eindimensionales Array verwendet, mit dessen Hilfe Task0 dann den Berechnungsabschnitt T_k^k ausführen kann. Zunächst wird in Task0 zur endgültigen Bestimmung von a_{kk} das k-te Element des globalen Arrays zu a_{kk} hinzuaddiert. Anschließend müssen n-k Elemente dieses globalen Arrays zu den a_{jk} beim Formen der k-ten Spalte der Dreiecksmatrix L hinzuaddiert werden. Für die nächste Iteration ist dann noch eine Initialisierung der n-k Elemente dieses Arrays notwendig. Diese zusätzliche Arbeit alleine hätte aufgrund der vektoriellen Ausführung keinen signifikanten Einfluß auf die parallele Ausführungszeit. Da jedoch während dieser Zeit Task1 blockiert ist, kann sie nicht zur parallelen Ausführung beitragen. Untersuchungen haben gezeigt, daß diese zusätzlich geleistete Arbeit und das dadurch bedingte Warten von Task1 für N = 600 pro Iteration durchschnittlich 1200 Takte ausmacht.

Dieser zusätzliche Aufwand beeinflußt die Höhe des insgesamt entstehenden Overheads. Die Höhe des Overheads für die Verwendung der EVENT-Routinen konnte dadurch minimiert werden, daß Task1 in jeder k-ten Iteration im Berechnungsabschnitt T_k^i etwas weniger Arbeit zu leisten hat als Task0. Dies hat zur Folge, daß CALL EVPOST(EVENT2) meistens vor CALL EVWAIT(EVENT2) ausgeführt wird. Durch diese etwas ungleichmäßige Verteilung der Arbeit konnten beispielsweise für N = 600 pro Iteration durchschnittlich 1800 Takte gespart werden. Der Overhead am Interaktionspunkt von EVENT1 lag hier pro Iteration durchschnittlich bei 4100 Takten und für EVENT2 bei 1600 Takten. Zu diesem Overhead müssen noch für jede Iteration Bankkonflikte hinzugerechnet werden.

Aus den Überlegungen für die Berechnungsfolgen der Cholesky-Reduktion ergab sich, daß das Formen aller k Spalten der unteren Dreiecksmatrix L sequentiell ausgeführt werden muß. Die dafür benötigte Zeit fließt in die Ermittlung des Multitasking-Teils f ein. Dazu kommt noch die Zeit, die für die sequentielle Ausführung bei der Lösung der beiden Gleichungssysteme $Ly = b$ und $L^t x = y$ erforderlich ist. Damit ergibt sich beispielsweise für N = 50 ein Multitasking-Teil von 83.1% und für N = 600 ein Multitasking-Teil von 98.1%.

Tabelle 8 gibt Aufschluß über die Ausführungszeiten, die die sequentielle bzw. parallele Cholesky-Reduktion für verschieden große Matrizen benötigten. In diesen Zeiten ist auch die für die Ermittlung des Lösungsvektors x benötigte Zeit enthalten. Daneben zeigt sie den tatsächlich erreichten und den theoretischen Speedup.

Tabelle 8. Ausführungszeiten für die sequentielle und parallele Cholesky-Reduktion der Matrix A einschließlich der Lösung des Gleichungssytems Ax = b.

Größe der Matrix A N × N	ohne Multitasking T_1/s	mit Multitasking T_p/s	theoretischer Speedup S_{th}	erreichter Speedup T_1/T_p
50	0.00231	0.00382	1.71	0.60
100	0.00958	0.00942	1.82	1.02
200	0.04592	0.03248	1.89	1.41
300	0.12357	0.07842	1.93	1.58
400	0.26225	0.15782	1.94	1.66
500	0.46239	0.27325	1.95	1.69
600	0.75489	0.44100	1.96	1.71

Abb. 19 zeigt den Verlauf des tatsächlich erreichten und des theoretischen Speedups. Der Unterschied zwischen theoretischem und erreichtem Speedup macht deutlich, daß für Gleichungssysteme kleiner Dimension der Overhead der Bibliotheksroutinen zu hoch ist.

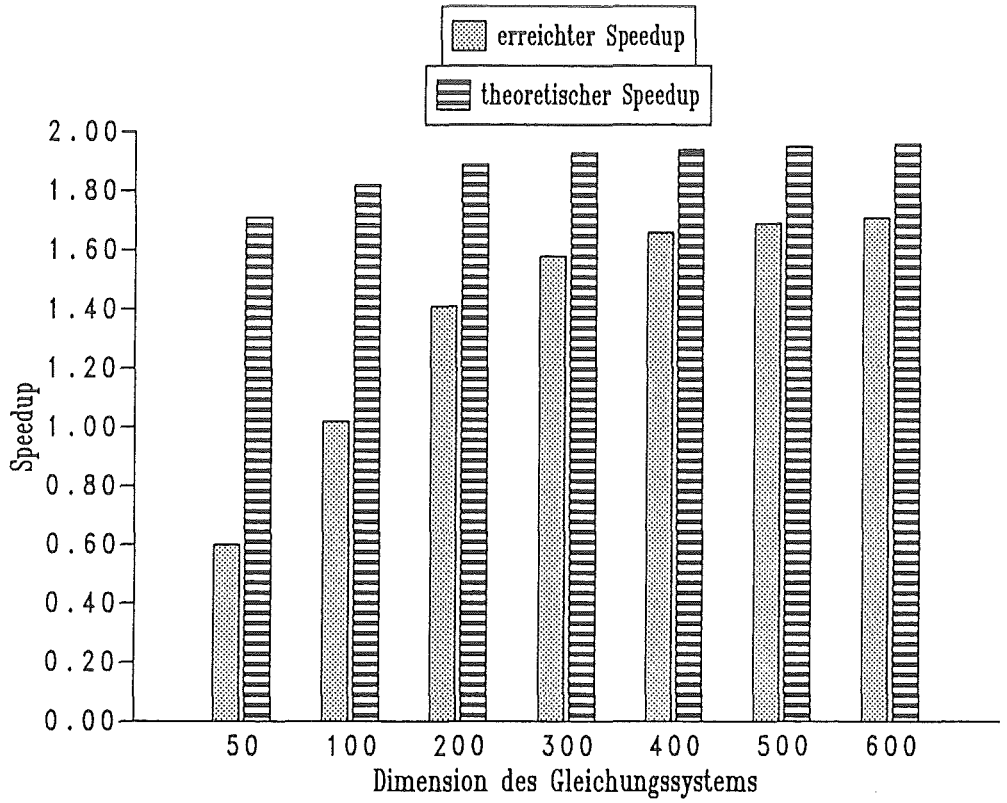


Abb. 19. Theoretische und erreichte Speedup-Werte der Cholesky-Reduktion einschl. Lösung des Gleichungssystems Ax = b: variable Dimension des Gleichungssystems; die Elemente der Matrix A und und des Vektors b wurden durch Zufallszahlen erzeugt.

Der Algorithmus zur Lösung des Gleichnungssystems Ax = b nach Reduktion der Matrix A mit Hilfe des Verfahrens von Cholesky ließe sich genauso parallelisieren wie der Algorithmus zur Lösung von Ax = b bei gegebener LU-Faktorisierung der Matrix A. Da jedoch auch hier die Granularität der Abschnitte, in denen die y_i und x_i berechnet werden, ähnlich klein im Verhältnis zu dem durch die Synchronisation entstehenden Overhead ist, kann für diesen Algorithmus durch Ausnutzung der Multitaskig-Fähigkeiten keine Beschleunigung erzielt werden.

5.0 SCHLUßBEMERKUNGEN

In dieser Arbeit wurden allgemeine Konzepte zur Steuerung der Kommunikation und Synchronisation paralleler Prozesse vorgestellt. Anhand des Konzepts des Multitasking der Firma CRAY RESEARCH konnten praktische Untersuchungen bez. der parallelen Ausführung von Unterprogrammen durchgeführt werden. Ziel dieser Arbeit war es, durch Nutzung der Multitasking-Fähigkeiten die Beschleunigungsmöglichkeiten einiger beispielhaft ausgewählter Standardalgorithmen zu untersuchen. Diese Untersuchungen haben gezeigt, daß eine befriedigende Beschleunigung nur dann erzielt werden kann, wenn die Granularität der Berechnungen wesentlich größer ist als der entstehende Overhead. Eine Granularität von 100 ms kann als ausreichend groß angesehen werden. Für eine befriedigende Beschleunigung muß der enstehende Overhead allerdings deutlich unter diesem Wert liegen. Dies machte sich besonders bei den Algorithmen zur Lösung linearer Gleichungssysteme bemerkbar.
Die Höhe des insgesamt entstehenden Overheads wurde vor allem bei den Algorithmen, die auf Matrixmethoden basieren, durch auftretende Speicherbankkonflikte beeinflußt. Dies konnte beim Jacobi-Iterationsverfahren verdeutlicht werden.

Die Ausführungszeiten der sequentiellen und parallelen Algorithmen waren teilweise davon abhängig, wie die Datenbereiche angelegt wurden. Bei der LU-Faktorisierung hat sich beispielsweise gezeigt, daß sich die Vereinbarung der globalen Matrix A im COMMON-Bereich für den sequentiellen Algorithmus in Bezug auf die Ausführungszeit wesentlich günstiger auswirkte als für den parallelen Algorithmus. Für den parallelen Algorithmus erwies sich die Parameterübergabe der globalen Matrix A als günstiger.

Bei der Implementierung der parallelen Algorithmen wurde festgestellt, daß sich eine etwas ungleichmäßige Verteilung der Arbeit wesentlich vorteilhafter auf die Höhe des Overheads auswirkt als eine vollständig balancierte Verteilung. Durch diese Umstellung konnte der enstehende Overhead bei den synchronen parallelen Algorithmen deutlich verringert werden. Bei einer vollständig balancierten Verteilung wäre der Overhead der Bibliotheksroutinen zu teuer gewesen. Obwohl der Vorteil der un-

gleichmäßigen Verteilung der Arbeit auch am Beispiel des Minimum-Maximum-Algorithmus genutzt werden konnte, kann man davon ausgehen, daß es für asynchrone parallele Algorithmen i. allg. nicht möglich sein wird, den Overhead durch diese Umstellung zu minimieren.

Das "timer trace" von J. L. Larson [CRAY,222] stellt einen hilfreichen Ansatz zur dynamischen Analyse des Ausführungsflusses eines Multitasking-Programms dar und war beispielsweise bei den Untersuchungen, wie die zu leistende Arbeit am günstigsten auf die Tasks zu verteilen ist, ein nützliches Hilfsmittel. Trotzdem können für kleine Problemgrößen durch seine Verwendung keine genauen Erkenntnisse über Granularitäten von Berechnungsabschnitten oder Overhead-Kosten gewonnen werden. Dies liegt daran, daß in der Implementierung selbst wieder LOCK-Routinen verwendet werden mußten, die die Ergebnisse verfälschen können. Wünschenswert wäre deshalb ein genaueres, auf Hardware basierendes Hilfsmittel zur dynamischen Analyse eines Multitasking-Programms.

Die Multitasking-Fähigkeiten werden vom Bibliothek-Scheduler bereitgestellt. Dieser stellt ein äußerst flexibles Konzept zur Verwaltung und Synchronisation paralleler Tasks dar. Der Bibliothek-Scheduler steuert die Interaktionen paralleler Tasks hauptsächlich über Software und verwendet zur Synchronisation der Tasks zumeist keine Semaphorregister. Dadurch kann busy-waiting vermieden werden. Für Probleme kleiner Granularität haben die Routinen des Bibliothek-Schedulers jedoch den Nachteil eines zu hohen Overheads.
Das Konzept des **Microtasking** der Firma CRAY RESEARCH verwendet zur Synchronisation paralleler Tasks direkt die Semaphorregister [Rein,85]. Dies hat den Vorteil eines äußerst geringen Overheads, der im Mittel bei 30 Takten liegt. Microtasking gestattet die Ausnutzung des Parallelismus auf Anweisungs-Ebene durch parallele Ausführung von DO-Schleifen, wodurch kleine Granularitäten zwischen den Interaktionspunkten möglich sein werden. Ziel weiterer Untersuchungen wird es deshalb sein, festzustellen, wie sich das Verhältnis zwischen kleiner Granularität und geringem Overhead auf die Beschleunigung eines Algorithmus durch Nutzung der Microtasking-Fähigkeiten auswirken wird.

[AnSch,83] Andrews, G. R. und Schneider, F. B.
Concepts and Notations for Concurrent Programming
Computing Surveys, Vol.15, Nr.1, März 1983, pp. 3-43

[Ari,82] Ben-Ari, M.
Principles of Concurrent Programming
Prentice-Hall, Inc., Englewood Cliffs, N. J., 1982

[BrHa,77a] Brinch Hansen, P.
Betriebssysteme
Carl Hanser Verlag, München, Wien, 1977

[BrHa,77b] Brinch Hansen, P.
The Architecture of Concurrent Programs
Prentice-Hall, Inc., Englewood Cliffs, N. J., 1977

[BrHa,78] Brinch Hansen, P.
Distributed Processes: A Concurrent Programming Concept
Communications of the ACM, Vol.21, Nr.11, November 1978,
pp. 934-941

[BrHa,81] Brinch Hansen, P.
Edison - a Multiprocessor Language
Software - Practice and Experience, Vol.11, Nr.4,
April 1981, pp. 325-361

[CES,71] Coffman jr., E. G., Elphick, J. und Shoshani, A.
System Deadlocks
Computing Surveys, Vol.3, Nr.2, Juni 1971, pp. 67-78

[CRAY,32] CRAY X-MP SERIES: Models 22 & 24
Mainframe Reference Manual
Revision A
CRAY-Research, Inc., July 1984, HR-0032

[CRAY,222] CRAY COMPUTER SYSTEMS: Technical Note
Multitasking User Guide
Revision A
CRAY-Research, Inc., January 1985, SN-0222

[Dijk,75] Dijkstra, E. W.
Guarded Commands, Nondeterminacy and Formal Derivation of Programs
Communications of the ACM, Vol.18, Nr.8, August 1975, pp. 453-457

[DoEi,84] Dongarra J. J. und Eisenstatt S. C.
Squeezing the Most out of an Algorithm in CRAY FORTRAN
ACM Transactions on Mathematical Software, Vol.10, Nr.3, September 1984, pp. 219-230

[GaPe,85] Gajski, D. D. und Peir, J.-K.
Essential Issues in Multiprocessor Systems
IEEE Computer, Vol.18, Nr.6, Juni 1985, pp. 9-27

[Geh,84] Gehani, Narian
ADA, An Advanced Introduction
Including Reference Manual for the ADA Programming Language
Prentice-Hall, Inc., Englewood Cliffs, N. J., 1984

[Hoa,78] Hoare, C. A. R.
Communicating Sequential Processes
Communications of the ACM, Vol.21, Nr.8, August 1978, pp. 666-677

[Holt,78] Holt, R. C., Graham, G. S., Lazowska, E. D. und Scott, M. A.
Structured Concurrent Programming with Operating System Applications
Addison-Wesley Publishing Company, 1978

[Hoß,81] Hoßfeld, F.
Parallele Algorithmen
Spezielle Berichte der KFA Jülich Nr. 125, 1981

[HwBr,84] Hwang, K. und Briggs, F. A.
Computer Architecture and Parallel Processing
McGraw-Hill Book Company, 1984

[KFA52,85] *Einführung in die Benutzung der CRAY*
Informationen für die Benutzer der zentralen Rechenanlagen
Kernforschungsanlage Jülich, 1985
KFA-ZAM-0052-CRAY

[KNU,75] Krayl, H., Neuhold, E. J., Unger, C.
Grundlagen der Betriebssysteme
Walter de Gryter & Co., 1975

[Kuck,78] Kuck, D. J.
The Structure of Computers and Computations
Volume One
John Wiley & Sons, Inc., 1978

[Kung,76] Kung, H. T.
Synchronized and Asynchronous Parallel Algorithms for Multiprocessors
In: Algorithms and Complexity, J. F. Traub (Hrsg.)
Academic Press, Inc., 1976

[Lamp,74] Lamport, L.
A New Solution of Dijkstra's Concurrent Programming Problem
Communications of the ACM, Vol.17, Nr.8, August 1974, pp. 453-455

[Lar,84a] Larson, J. L.
Practical Concerns in Multitasking on the CRAY X-MP
Workshop on Using Multiprocessors in Meteorological Models, December 3-6, 1984, ECMWF, Reading, England

[Lar,84b] Larson, J. L.
Multitasking on the Cray X-MP-2 Multiprocessor
IEEE Computer, Vol.17, Nr.7, Juli 1984, pp. 62-69

[LKK,83] Lord, R. E., Kowalik, J. S. und Kumar, S. P.
Solving Linear Algebraic Equations on an MIMD Computer
Journal of the ACM, Vol.30, Nr.1, Januar 1983, pp. 103-117

[NeRi,85] Nelson, L. und Rigsbee, P.
Multitasking at Cray
CRAY CHANNELS, Summer 1985, pp. 12-15

[Num,79] Locher, F., Jetter, K., Möller, M., Sauter, W. und Weber, E.
Numerische Mathematik I
Kurseinheit 5: Iterative Verfahren zur Lösung von linearen Gleichungssystemen
Fernuniversität Hagen, Fachbereich Mathematik, 1979

[PeSi,85] Peterson, J. L. und Silberschatz, A.
Operating System Concepts
Second Edition
Addison-Wesley Publishing Company, 1985

[Pet,81] Peterson, G. L.
Myths about the Mutual Exclusion Problem
Information Processing Letters, Vol.12, Nr.3, Juni 1981,
pp. 115-116

[Rein,85] Reinhardt, S.
A Data-Flow Approach to Multitasking on
CRAY X-MP Computers
Proceedings of the Tenth ACM Symposium on Operating
Systems Principles, Dezember 1985
ACM Operating Systems Review, Vol.19, Nr.5, 1985
pp. 107-114

[RiTh,74] Ritchie, D. M. und Thompson, K.
The UNIX Time-Sharing System
Communications of the ACM, Vol.17, Nr.7, Juli 1974,
pp. 365-375

[Sedg,78] Sedgewick, R.
Implementing Quicksort Programs
Communications of the ACM, Vol.21, Nr.10, Oktober 1978,
pp. 847-857

[Sedg,83] Sedgewick, R.
Algorithms
Addison-Wesley Publishing Company, 1983

[Stew,73] Stewart, G. W.
Introduction to Matrix Computations
Academic Press, Inc., 1973

[Weck,82] Weck, G.
Prinzipien und Realisierung von Betriebssystemen
Teubner Studienbücher Informatik
B. G. Teubner, Stuttgart, 1982

[Wijn,75] van Wijngarden, A., Mailloux, B. J., Peck, J. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T. und Fisker, R. G.
Revised Report on the Algorithm Language ALGOL68
Acta Informatica, Vol.5, Nr.1-3, 1975

[Wirth,79] Wirth, N.
Algorithmen und Datenstrukturen
Teubner Studienbücher Informatik
B. G. Teubner, Stuttgart, 1979

[Zima,80] Zima, H.
Betriebssysteme. Parallele Prozesse
Reihe Informatik, Bd. 20
Bibliographisches Institut, 1980